UTILIZING QUERY LOGS FOR DATA REPLICATION AND PLACEMENT IN BIG DATA APPLICATIONS

A DISSERTATION SUBMITTED TO
THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Ata Türk
September, 2012

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Prof. Dr. Cevdet Aykanat(Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Prof. Dr. Fazlı Can

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Prof. Dr. Enis Çetin

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Assist. Prof. Dr. Murat Manguoğlu

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.

Assoc. Prof. Dr. Hakan Ferhatosmanoğlu

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

UTILIZING QUERY LOGS FOR DATA REPLICATION AND PLACEMENT IN BIG DATA APPLICATIONS

Ata Türk
Ph.D. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
September, 2012

The growth in the amount of data in todays computing problems and the level of parallelism dictated by the large-scale computing economics necessitates high-level parallelism for many applications. This parallelism is generally achieved via data-parallel solutions that require effective data clustering (partitioning) or declustering schemes (depending on the application requirements). In addition to data partitioning/declustering, data replication, which is used for data availability and increased performance, has also become an inherent feature of many applications. The data partitioning/declustering and data replication problems are generally addressed separately. This thesis is centered around the idea of performing data replication and data partitioning/declustering simultenously to obtain replicated data distributions that yield better parallelism. To this end, we utilize query-logs to propose replicated data distribution solutions and extend the well known Fiduccia-Mattheyses (FM) iterative improvement algorithm so that it can be used to generate replicated partitioning/declustering of data. For the replicated declustering problem, we propose a novel replicated declustering scheme that utilizes query logs to improve the performance of a parallel database system. We also extend our replicated declustering scheme and propose a novel replicated re-declustering scheme such that in the face of drastic query pattern changes or server additions/removals from the parallel database system, new declustering solutions that require low migration overheads can be computed. For the replicated partitioning problem, we show how to utilize an effective single-phase replicated partitioning solution in two well-known applications (keyword-based search and Twitter). For these applications, we provide the algorithmic solutions we had to devise for solving the problems that replication brings, the engineering decisions we made so as to obtain the greatest benefits from the proposed data distribution, and the implementation details for realistic systems. Obtained results indicate that utilizing query-logs and performing

replication and partitioning/declustering in a single phase improves parallel performance.

Keywords: replication, iterative improvement, query-log-aware, partitioning, declustering.

ÖZET

SORGU GÜNLÜKLERİ KULLANARAK VERİ ÇOKLAMA VE YERLEŞTİRME PROBLEMLERİNİN ÇÖZÜMÜ

Ata Türk
Bilgisayar Mühendisliği, Doktora
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Eylul, 2012

Günümüz hesaplama sistemlerinin işlemesi gereken veri miktarlarındakı artış ve hesaplama sistemleri altyapı ve ekonomileri sebepleri ile uygulamaların çoğunda yüksek seviyede paralleleştirme gerekmektedir. Bu paralleleştirme genellikle veri-paralel çözümlerle gerçekleştirilir ki bu çözümler de efektif veri gruplama (partitioning) ve veri dağıtma (declustering) yöntemleri gerektirir. Veri gruplama ve dağıtma yöntemlerinin yanında, gerek kullanılabilirliği gerekse performansı arttırma adına veri çoklama yöntemleri de sıkça kullanılmaya başlanmıştır. Veri bölümleme ya da dağıtma ve veri çoklama problemleri genellikle iki farklı aşamada çözümlenmeye çalışılırlar. Bu tezdeki çalışmalar, veri bölümleme/dağıtma ve veri çoklama problemlerinin tek bir aşamada yapılması sureti ile daha etkin çoklanarak bölümlenmiş/dağıtılmiş sistemler elde edilmesi fikri üzerine yoğunlaşmıştır. Bu amaçla, bölümleme sistemlerinde yaygın olarak kullanılan Fiduccia-Mattheyses (FM) yinelemeli iyileştirme algoritması çoklama işlemini de kapsayacak şekilde genişletilmiştir. Bu algoritma kullanılarak sorgu günlükleri kullanan veri tabanı uygulamalarının performansını arttıracak bir çoklamalı veri dağıtma sistemi önerilmiştir. Ayrıca bu çoklamalı veri dağıtma sisteminin sorgu desenlerinde değişimler, yeni sunucu ekleme ya da çıkarma işlemleri gibi durumlar karşısında mümkün olduğunca az veri taşımaşı yaparak kendini adapte etmesini sağlayan genişletme ve ilaveler önerilmiştir. Daha sonra, çoklamalı bölümleme problemi için geliştirilen tek-aşamalı çoklamalı bölümleme aracı, yaygın olarak bilinen iki uygulama (kelime bazlı arama ve Twitter) üzerinde test edilmiştir. Elde edilen sonuçlar sorgu günlükleri kullanımının ve çoklama ile veri bölümleme/dağıtma işlemlerinin tek aşamada yapılmasının parallel performansı arttırdıgını göstermektedir.

Anahtar sözcükler: çoklama, yinelemeli ilerleme, sorgu-geçmişi-bilinçli,

bölümleme, dağıtma.

Acknowledgement

First and foremost I would like to thank to my advisor Prof. Cevdet Aykanat. He has been like a second father to me, not only directing me out of dark corners and alleys of academic life to bright sunshine, but also giving me support and advice in all aspects of everyday life, not sparing an ounce of life experience and knowledge whenever he could. I think he is a true "teacher", the embodiment of all the meaning behind the word, and I say that believing there is not a greater honorific for a man than that. (I would also like to thank the family of Prof. Aykanat for sharing so much time of their father and husband with us pesky grads and not resenting it.)

I would like to thank my family for all their love and encouragement. Especially to my mother for supporting me in all possible ways to pursue a PhD title. I'd like to thank to my brother Buğra just for being my brother, really, just for being. Our bond has always been something to draw strength for me and always will be.

I would like to thank to my colleague Reha Oğuz Selvitopi for his invaluable contributions to some of the works presented in this thesis. His intelligent mind and hard-working kindled a stronger belief and desire for academic studies in me.

I would like to thank my good friend Kamer Kaya for being a rock in my life, someone with whom I know I can share all my problems, how embarrassing or troublesome they may be. Knowing that is great even when you don't share much. It is that knowledge and trust that gives infinite comfort.

I would like to thank all my friends in Bilkent University and especially my (old and new) colleagues from Parallel Computing Group. Special thanks to Berkant Barla Cambazoglu, for being a second advisor for me, not only tutoring me in academics but also in life philosophy.

And most of all my loving, supportive, encouraging, and patient wife Esra, who changed my life and myself for good and made me whole. I love you with

all the loving in the world and a little more. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last decade, the amount of data generated has simply exploded. Following the "digital revolution" of 1980s, the "mobile and social data revolution" of early 2000s resulted with the accumulation of unprecedented amounts of public data in the form of text, voice, image, and video and the associated miscellaneous such as web logs, query logs, search indexes, friendship graphs and GPS logs. Applications spanning and serving in more than a few continents have reshaped the design and functioning of data centers housing these applications. The scale of the problems encountered while serving these applications along with the trend change in chip design technologies have turned "good parallelism" from a "plus" first into "norm" and then into a "must" in application design.

In this thesis, our aim is to improve the parallel performance of large-scale applications by deciding on the placement and replication of data. Improved data placement and replication enables reduction of parallelization overhead through better computational load balancing and/or reduced communication overheads. We plan to focus on utilization of query logs collected by large-scale applications for this purpose. We believe that any application that generates responses to user queries must make use of its query logs in determining data placement and replication strategies.

Depending on the specifics and requirements of applications and the underlying parallel architecture utilized to serve the application, the placement and replication requirements may differ. For example, in an online analytical processing (OLAP) type of application, utilizing query logs to place related data together (clustering) to minimize communication overheads might be desirable, whereas in an I/O bound online transaction processing (OLTP) type of application, utilizing query logs to place related data separate from each other (declustering) to minimize individual query response times might be preferable.

In some applications, clustering and declustering might even need to be used in synchrony by grouping related data in certain application levels and distributing in certain other levels. Consider an I/O intensive application that spans a few datacenters. There are a number of different levels for the data placement and replication in such a framework: datacenter-level, rack-level and finally node-level. Arranging data placement so that response generation for each query can be performed within a single datacenter would reduce query response times significantly, since cross datacenter communication is quite costly. Similarly, within a datacenter, clustering related data into the same racks also makes sense, since generally, cross-rack communication is also costly [1]. On the other hand, within a datacenter rack, scattering the data items that are likely to be queried together to multiple nodes would produce better load balancing and would probably be faster than clustering them in a single node, since this enables parallelism in disk access costs. Furthermore, generally in-rack network bandwidth is high and for a node, accessing the disk of another node in the same rack is not much costlier than accessing its own disk [1]. Add to these perplexing clustering and declustering requirements the necessity for replication and the desire to utilize replication for improving performance and you end up with a difficult data replication and placement problem, i.e., the problem of deciding which data items to replicate and where to place those data items. Furthermore, in certain applications, even decisions related with which replica to use in answering queries affects system performance, thus, replica selection problem, which is tightly coupled with data placement and replication problem may need to be addressed.

In this thesis we separately show how solutions to replicated declustering and

replicated clustering problems can improve the performance of parallel applications. Interestingly, our solutions to both the replicated clustering and replicated declustering problems are developed by extending the main ideas of the same algorithm: The Fiduccia-Mattheyses (FM) heuristic. FM heuristic [2], originally proposed for efficiently bipartitioning circuit hypergraphs, is a linear-time iterative improvement algorithm that is utilized effectively in graph and hypergraph partitioning. FM-like iterative improvement algorithms also found use in many different areas such as declustering schemes [3, 4]. One of the main contributions of this thesis is to propose FM-like iterative improvement heuristics both for the replicated declustering and the replicated clustering (partitioning) problems. FM algorithm classically supports only move-based neighborhood definitions and iterates over the solution space using this operation. Our replicated FM algorithms extend the neighborhood definitions of the FM algorithm to include replication and unreplication operations. Extending FM heuristics to support replica placement opens an avenue of possibilities in various areas and in this thesis we discuss only four of such studies where this extended heuristic is put to good use.

In Chapter 2, we first propose a query-log aware FM-like replicated iterative improvement heuristic for the two-way declustering problem. We also provide simple closed-form expressions for computing the cost of a query in a two-way replicated declustering. Utilizing these expressions, we avoid usage of expensive network-flow based algorithms for the construction of optimal query schedules. By recursively applying our two-way replicated declustering algorithm we obtain a K-way replicated declustering. Our unreplication algorithm prevents unnecessary replications to advance to the next levels in the recursive framework. We then propose an efficient multi-way replicated refinement heuristic that considerably improves the obtained K-way replicated declustering via multi-way move and multi-way replication operations. In this iterative algorithm, we adapt a novel idea about multi-way move operations and obtain an efficient greedy multi-way move/replication scheme that enables us to avoid many of the complexities due to the increase in the number of servers. We also present an efficient scheme to avoid the necessity of computing the optimal schedules of all queries at each iteration of our multi-way refinement algorithm. The proposed scheme enables us

to compute the optimal schedules of all queries just once, at the beginning of the multi-way refinement, and then update the schedules incrementally according to the performed operations.

In Chapter 3, we enhance our replicated declustering scheme so as to make it more scalable, a most desirable property for large-scale applications. This proposed re-declustering scheme supports fast and effective computation of new declustering solutions under serious query pattern changes, server additions or server removals. To this end, we first propose a novel weighted bipartite matching model that given a newly proposed replicated declustering solution for a dataset, and an existing replicated mapping of the data items in the dataset to the servers of a parallel database system, optimally computes a matching of the overlapping subsets in the replicated declustering solution to servers such that the number of data item migrations necessary to realize the new solution is minimized. We then propose a novel abstraction scheme that enables the encoding of migration operations necessary to realize the new solution as queries. This abstraction enables the use of a our K-way replicated declustering scheme of Chapter 2 for the solution of re-declustering problem with minor extensions. All in all, we propose a three-phase log-utilizing replicated re-declustering scheme that strikes a balance between the objectives of query cost and migration cost reductions in Chapter 3.

In a work which is not presented in this thesis ([5]), we had shown how our ideas in Chapter 2 could be applied to the replicated clustering problem. [5] also presents how to utilize these ideas in a multi-level framework and presents the multi-level replicated clustering tool `rpPaToH`, which can be used for replicated hypergraph partitioning. In this thesis, we show how two well-known applications, namely keyword-based query processing and Tweeter can benefit from the use of log-aware replicated clustering (Chapters 4 and 5 respectively).

More specifically, in Chapter 4, we implement a successful parallel keyword-based query processing system that utilizes a replicated inverted index and we provide details of our implementation and the reasons behind our design choices.

We show how our index replication approach based on replicated hypergraph partitioning [5] outperforms classic replication approaches. We also propose various heuristic solutions for the replica selection problem in query processing.

On the other hand, in Chapter 5, we consider the simplest of online social networks, Twitter, which serves tweet reading and tweeting functionalities over the intrinsic social relations of its users. We propose a novel temporal activity hypergraph model to represent the multi-way relations between user actions in Twitter, which we partition via `rpPaToH` and replicate and distribute the user data according to the result of this replicated partitioning. We compare the results of our replicated partitioning scheme with circular replication schemes used in distributed hash tables to show that the proposed scheme can greatly increase locality.

In Chapter 6, we conclude with a brief listing of the accomplishments of this thesis and discussions of the obtained results. We also note that two of the four studies discussed in this thesis are works on replicated declustering, whereas the other two are works on replicated clustering and we briefly discuss how these two antipodal problems (the declustering and partitioning problems) with conflicting objectives can arise together in todays big-data computing settings, leaving the meshing of two solutions as future work.

Chapter 2

Query-Log Aware Replicated Declustering

Data declustering and replication can be used to reduce I/O times related with processing of data intensive queries. Declustering parallelizes the query retrieval process by distributing the data items requested by queries among several disks. Replication enables alternative disk choices for individual disk items and thus provides better query parallelism options. In general, existing replicated declustering schemes do not consider query log information and try to optimize all possible queries for a specific query type, such as range or spatial queries. In such schemes, it is assumed that two or more copies of all data items are to be generated and scheduling of these copies to disks are discussed. However, in some applications, generation of even two copies of all of the data items is not feasible, since data items tend to have very large sizes. In this chapter we assume that there is a given limit on disk capacities and thus on replication amounts. We utilize existing query-log information to propose a selective replicated declustering scheme, in which we select the data items to be replicated and decide on their scheduling onto disks while respecting disk capacities. We propose and implement an iterative improvement algorithm to obtain a two-way replicated declustering and use this algorithm in a recursive framework to generate a multi-way replicated declustering. Then we improve the obtained multi-way replicated declustering by

efficient refinement heuristics. Experiments conducted on realistic datasets show that the proposed scheme yields better performance results compared to existing replicated declustering schemes.

2.1 Introduction

2.1.1 Related work

Data declustering is a data scattering technique used in parallel-disk architectures to improve query response time performances of I/O intensive applications. The aim in declustering is to optimize the processing time of each query requested from a parallel-disk architecture. This is achieved by reducing the number of disk accesses performed by a single disk of the architecture while answering a single query. Declustering has been shown to be an NP-complete problem in some contexts [6], [4].

Declustering is widely investigated in applications where large spatial data are queried. In such applications, queries are in the form of ranges requesting neighboring data points, and hence, related declustering schemes try to scatter neighboring data items into separate disks instead of exploiting query log information. For a good survey of declustering schemes optimized for range queries see [7] and the citations within.

There are some applications that also query very large data items in a random fashion and in such applications utilization of query log information is of essence for efficient declustering [6], [4], [3]. In [6], the declustering problem with a given query distribution is modeled as a max-cut partitioning of a weighted similarity graph, where data items are represented by vertices and an edge between two vertices implies that corresponding data items appear in at least one common query. In [4] and [3], the deficiencies of the weighted similarity graph model are addressed and hypergraph models which encode the total I/O cost correctly are proposed.

Data replication is a widely applied technique in various application areas such as distributed data management [8] and information retrieval [9, 10] to achieve fault tolerance and fault recovery. Data replication can also be exploited to achieve higher I/O parallelism in a declustering system [11]. However, while performing replication, one has to be careful about consistency considerations, which arise in update and delete operations. Furthermore, write operations tend to slow down when there is replication. Finally, replication means extra storage requirement and there are applications with very large data sizes where even two-copy replication is not feasible. Thus, if possible, unnecessary replication has to be avoided and techniques that enable replication under given size constraints must be studied.

When there is data replication, the problem of query scheduling has to be addressed as well. That is, when a query arrives, we have to decide which replicas will be used to answer the query. A maximum-flow formulation is proposed in [12] to solve this scheduling problem optimally. There are replicated declustering schemes that aim to minimize this scheduling overhead [13, 14], while minimizing I/O costs. A variation of this problem arises when replicas are assumed to be distributed over different sites, where each site hosts a parallel-disk architecture [15]. This variation can be modeled as a maximum flow problem as well.

Most of the existing replicated declustering schemes proposed for range queries are discussed in [16, 17]. There are some replicated declustering schemes proposed for arbitrary queries as well ([18], [19]). All of these schemes ([16], [17], [18], [19]) assume items with equal sizes and they also assume that all data items will be requested equally likely and thus generate equal number of replicas for all data items. Furthermore, they replicate all data items two or more times.

In [18], Random Duplicate Assignment (RDA) scheme is proposed. RDA stores a data item on two disks chosen randomly from the set of disks and it is shown that the retrieval cost of random allocation is at most one more than the optimal cost with high probability (when there are at least two-copies of all data items). In [15, 20], Orthogonal Assignment (OA) is proposed. OA is a two-copy

replication scheme for arbitrary queries and if the two disks that a data item is replicated at are considered as a pair, each pair appears only once in the disk allocation of OA. In [19], Design Theoretic Assignment (DTA) is proposed. DTA uses the blocks of a $(K, c, 1)$ design for c-copy replicated declustering using K disks. A block and its rotations can be used to determine the disks on which the data items are stored. Even though both OA and DTA can be modified to achieve selective replication, they do not utilize query log information. However, with the increasing usage in GIS and spatial database systems, such information is becoming highly available, and it is desirable for a replication scheme to be able to utilize this information. A simple motivating example for utilizing query-logs can be found in Section 2.3.

2.1.2 Contributions

In this work we present a selective and query-log aware replication scheme which works in conjunction with declustering. The proposed scheme utilizes the query log information to minimize the aggregate parallel query response time while obeying given replication constraints due to disk sizes. There are no restrictions on the replication counts of individual data items. That is, some data items may be replicated more than once while some other data items may not even be replicated at all.

We first propose an iterative-improvement-based replicated two-way declustering algorithm. In this algorithm, in addition to the replication operation that we proposed in [21], we successfully incorporate unreplication operation to the replicated two-way declustering algorithm to prevent unnecessary replications. We also provide simple closed-form expressions for computing the cost of a query in a two-way replicated declustering. Utilizing these expressions, we avoid usage of expensive network-flow based algorithms for the construction of optimal query schedules. By recursively applying our two-way replicated declustering algorithm we obtain a K-way replicated declustering. Our unreplication algorithm prevents unnecessary replications to advance to the next levels in the recursive framework.

We then propose an efficient multi-way replicated refinement heuristic that considerably improves the obtained K-way replicated declustering via multi-way move and multi-way replication operations. In this iterative algorithm, we adapt a novel idea about multi-way move operations and obtain an efficient greedy multi-way move/replication scheme. We also present an efficient scheme to avoid the necessity of computing the optimal schedules of all queries at each iteration of our multi-way refinement algorithm. The proposed scheme enables us to compute the optimal schedules of all queries just once, at the beginning of the multi-way refinement, and then update the schedules incrementally according to the performed operations.

The rest of the chapter is organized as follows. Section 2.2 presents the notation and the definition of the problem. A motivating example is provided in Section 2.3 and the proposed scheme is presented in Section 2.4. In Section 2.5, a running example demonstrating move, replication and unreplication gain updates can be found. The details of the algorithms used in the proposed replicated declustering scheme are presented in Section 2.6. In Section 2.7, detailed complexity analyses of the recursive replicated declustering and multi-way replicated refinement phases of our algorithm are presented. Section 2.8, contains experiments and comparisons of the proposed approaches with two state-of-the-art replications schemes.

2.2 Notation and Definitions

We are given a dataset $\mathcal{D}$ with $|\mathcal{D}|$ indivisible data items and a query set Q with $|Q|$ queries, where a query $q \in Q$ requests a subset of data items, i.e., $q \subseteq \mathcal{D}$. Each data item $d \in \mathcal{D}$ can represent a spatial object, a multi-dimensional vector or a cluster of data records depending on the application. $s(d)$ indicates the storage requirement for d and $s(\mathcal{D}') = \sum_{d \in \mathcal{D}'} s(d)$ indicates the storage requirement for data subset $\mathcal{D}'$. Query information can be extracted by either application usage prediction or mining existing query logs, with the assumption that future queries will be similar to older ones. In a few applications, it is more appropriate to

Table 2.1: The notations used in this chapter

Symbol	Description
$\mathcal{D}$	Dataset
Q	Query set
K	Total number of disks
$\mathcal{D}_k$	Set of data items assigned to disk k
C_{max}	Maximum storage capacity of a disk
d_i	A data item in the dataset
$s(d_i)$	Storage requirement for data item d_i
q	A query in the query set
$\|q\|$	Number of data items requested by q
$f(q)$	Frequency of q in Q
$r(q)$	Response time for q
$t_k(q)$	Response time of disk k for q
$S_{opt}(q)$	Optimal scheduling for q
$r_{opt}(q)$	Optimal response time for q
R_K	A K-way replicated declustering
$Tr(R_K, Q)$	Parallel response time of R_K for Q
$Tr_{opt}(Q)$	Optimal parallel response time for Q
$TrO(R_K, Q)$	Parallel response time overhead of R_K for Q
$g_m(d)$	In a two-way replicated declustering phase, reduction to be observed in the overall query processing cost, if d is moved to the other disk.
$g_r(d)$	In a two-way replicated declustering phase, reduction to be observed in the overall query processing cost, if d is replicated in both disks.
$g_{u_X}(d)$	In a two-way replicated declustering phase, reduction to be observed in the overall query processing cost, if a replica of d is deleted from disk $\mathcal{D}_X$.
$vg(d)$	Number of queries requesting d such that the disk(s) that d resides in serve(s) more than optimal number of data items for these queries.
$g_m(d, k)$	In a K-way replicated declustering phase, reduction to be observed in the overall query processing cost, if d is moved to disk k.
$g_r(d, k)$	In a K-way replicated declustering phase, reduction to be observed in the overall query processing cost, if d is replicated in disk k.

apply declustering such that items that have common features are stored on separate disks [22, 23, 24]. However, even in such applications, each query can be considered as a set of features and the discussions in the following sections still hold.

In a given query set Q, two data items are said to be neighbor if they are requested together by at least one query. Each query q is associated with a relative frequency $f(q)$ which indicates the probability that q will be requested. Query frequencies can be extracted from the query log. We assume that all disks

are homogeneous and the retrieval time of all data items on all disks are equal and can be accepted as one for practical purposes.

Definition K-Way Replicated Declustering: Given a set $\mathcal{D}$ of data items, K homogeneous disks with storage capacity C_{max}, and a maximum allowable replication ratio r, $R_K = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_K\}$ is said to be a K-way replicated declustering of $\mathcal{D}$, where $\mathcal{D}_k \subseteq \mathcal{D}$ for $1 \leq k \leq K$, $\cup_{k=1}^{K} \mathcal{D}_k = \mathcal{D}$, and R_K satisfies the following feasibility conditions for $1 \leq k \leq K$, when each decluster $\mathcal{D}_k$ is assigned to a separate disk:

- Disk capacity constraint: $s(\mathcal{D}_k) \leq C_{max}$
- Replication constraint: $\sum_{1 \leq k \leq K} s(\mathcal{D}_k) \leq (1 + r) \times s(\mathcal{D})$.

The optimal schedule for a query q minimizes the maximum number of data items requested from a disk for q. Given a replicated declustering R_K and a query q, an optimal schedule $S_{opt}(q)$ for q can be calculated by a network-flow based algorithm [12] in $O(|q|^2 \times K)$ time, if we assume homogeneous data item retrieval times. $S_{opt}(q)$ indicates which replicas of the data items will be accessed during processing q.

Definition Given a replicated declustering R_K, a query q and an optimal schedule $S_{opt}(q)$ for q, response time $r(q)$ for q is:

$$r(q) = \max_{1 \leq k \leq K} \{t_k(q)\}, \tag{2.1}$$

where $t_k(q)$ denotes the total retrieval time of data items from disk $\mathcal{D}_k$ that are requested by q. Under homogeneous data item retrieval times assumption, $t_k(q)$ can also be considered as the number of data items retrieved from $\mathcal{D}_k$ for q.

Definition The total parallel response time of a replicated declustering R_K for a query set Q is:

$$Tr(R_K, Q) = \sum_{q \in Q} f(q) r(q). \tag{2.2}$$

Definition A replicated declustering R_K is said to be *strictly optimal* for a query set Q iff it is optimal for every query $q \in Q$, i.e., $r(q) = r_{opt}(q), \forall\ q \in Q$, where

$$r_{opt}(q) = \lceil |q|/K \rceil. \tag{2.3}$$

Total parallel response time of a strictly optimal replicated declustering is called $Tr_{opt}(Q)$ and is:

$$Tr_{opt}(Q) = \sum_{q \in Q} f(q) r_{opt}(q). \tag{2.4}$$

Definition The total parallel response time overhead of a replicated declustering R_K for a query set Q is:

$$TrO(R_K, Q) = Tr(R_K, Q) - Tr_{opt}(Q). \tag{2.5}$$

Definition K-Way Replicated Declustering Problem: Given a set $\mathcal{D}$ of data items, a set Q of queries, K homogeneous disks each with a storage capacity of C_{max}, and a maximum allowable replication ratio r, find a K-way replicated declustering R_K of $\mathcal{D}$ that minimizes the total parallel response time $Tr(R_K, Q)$. Note that minimizing $Tr(R_K, Q)$ is equivalent to minimizing $TrO(R_K, Q)$, since $Tr_{opt}(Q)$ is a constant.

2.3 A motivating example for query-log aware replicated declustering

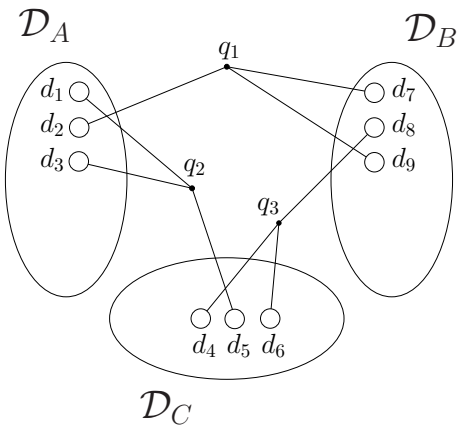
In this section we provide a motivating example for query-log aware replicated declustering. In Fig. 2.1(a), we have a dataset $\mathcal{D}$ and three common queries requested from this dataset. Fig. 2.1(b) shows a 2-copy replicated declustering of $\mathcal{D}$ performed by a replicated declustering algorithm that is unaware of query logs. As seen in Fig. 2.1(b), the costs of queries q_1, q_2, and q_3 are the same with their costs prior to replication, thus replication has no positive effect for these queries. However, as seen in Fig. 2.1(c), by selectively replicating data items to reduce the costs of common queries it is possible to reduce the overall cost by performing much less amount of replication.

2.4 Proposed Approach

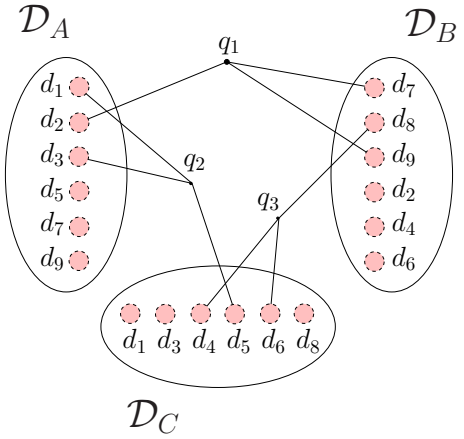
We propose a two-phase approach for solving the K-way replicated declustering problem. In the first phase, we use a recursive replicated declustering heuristic to obtain a K-way replicated declustering. We should note that, by allowing imbalanced two-way declusters in this phase, we are able to obtain K-way declusterings for arbitrary K values. In the second phase, we use a refinement heuristic to improve the K-way replicated declustering obtained in the first phase. In the following two subsections we provide the details of operations performed in these phases. A detailed complexity analysis of the recursive replicated declustering and multi-way replicated refinement phases will be presented in Section 2.7.

2.4.1 Recursive replicated declustering phase

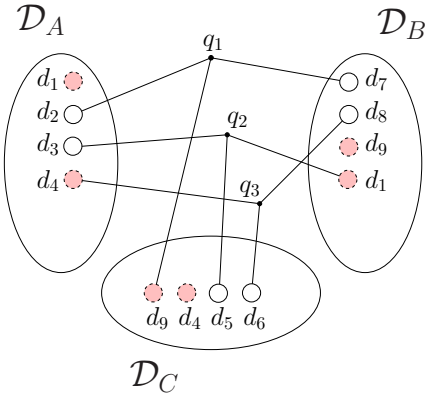
The objective in the recursive replicated declustering phase is to evenly distribute the data items of queries at each two-way replicated declustering step of the recursive framework. That is, at each two-way step, we try to attain optimal



(a) A 3-disk declustering of $\mathcal{D}$ with three common queries



(b) A 2-copy replicated declustering of $\mathcal{D}$



(c) A query-log aware replicated declustering of $\mathcal{D}$

Figure 2.1: A dataset $\mathcal{D} = \{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9\}$ and a 3-disk system, where $q_1 = \{d_1, d_3, d_5\}$, $q_2 = \{d_4, d_6, d_8\}$, $q_3 = \{d_2, d_7, d_9\}$ are three common queries.

response time $r_{opt}(q) = \lceil |q|/2 \rceil$ for each query q as much as possible. This objective is somewhat restrictive and it will not completely model the minimization of the objective function for the K-way replicated declustering problem. But it is expected to produce a "good" initial K-way replicated declustering for the multi-way refinement phase. The even query distribution obtained after the recursive replicated declustering phase is assumed to avoid a bad locally optimal declustering by providing flexibility in the search space of the multi-way refinement scheme.

2.4.1.1 Two-way replicated declustering

The core of our recursive replicated declustering algorithm is a two-way replicated declustering algorithm. In this algorithm, we start with a given (and possibly randomly generated) initial feasible two-way declustering of the dataset $\mathcal{D}$, say $R_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$, and iteratively improve R_2 by three refinement operations defined over the data items: Namely *move*, *replication* and *unreplication* operations. In order to perform these three operations we consider four different gain values for each data item d:

- move gain ($g_m(d)$): the reduction to be observed in the overall query processing cost, if d is moved to the other disk,
- replication gain ($g_r(d)$): the reduction to be observed in the overall query processing cost, if d is replicated to the other disk,
- unreplication-from-A gain ($g_{u_A}(d)$): the reduction to be observed in the overall query processing cost, if a replica of d is deleted from $\mathcal{D}_A$,
- unreplication-from-B gain ($g_{u_B}(d)$): the reduction to be observed in the overall query processing cost, if a replica of d is deleted from $\mathcal{D}_B$.

Unreplication gains are only meaningful for data items that are replicated. Similarly, in a two-way declustering, move and replication gains are only meaningful for data items that are not replicated. Thus, for any data item, only two gain values need to be maintained.

A two-way replicated declustering $R_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$ can be considered as partitioning the dataset $\mathcal{D}$ into three mutually disjoint parts: A, B, and AB, where part A is composed of the data items that are only stored in disk $\mathcal{D}_A$, part B is composed of the data items that are only stored in disk $\mathcal{D}_B$, and part AB is composed of the data items that are replicated. In this view,

$$\mathcal{D}_A = A \cup AB \quad \text{and} \quad \mathcal{D}_{\mathrm{B}} = \mathrm{B} \cup \mathrm{AB}. \tag{2.6}$$

A variable $State(d)$ is kept to store the part information of each data item d.

For each query q, we maintain a 3-tuple

$$dist(q) = (|q_A| : |q_B| : |q_{AB}|), \tag{2.7}$$

where $|q_A|$, $|q_B|$, and $|q_{AB}|$ indicate the number of data items of q in parts A, B, and AB, respectively. That is,

$$q_A = q \cap A \ , \quad \mathrm{q_B} = \mathrm{q} \cap \mathrm{B} \quad \text{and} \quad \mathrm{q_{AB}} = \mathrm{q} \cap \mathrm{AB}. \tag{2.8}$$

The total number of data items requested by query q is equal to: $|q| = |q_A| + |q_B| + |q_{AB}|$.

Using the above notation, the retrieval times of a given query q from disks $\mathcal{D}_A$ and $\mathcal{D}_B$ can be written as follows, without loss of generality assuming that $|q_A| \geq |q_B|$:

$$\begin{aligned} t_A(q) &= \begin{cases} \lceil |q|/2 \rceil & \text{if } |q_{AB}| \geq (|q_A| - |q_B|) - 1 \\ |q_A| & \text{otherwise} \end{cases} \\ t_B(q) &= \begin{cases} \lfloor |q|/2 \rfloor & \text{if } |q_{AB}| \geq (|q_A| - |q_B|) - 1 \\ |q_B| + |q_{AB}| & \text{otherwise} \end{cases} \end{aligned} \tag{2.9}$$

Here the "$|q_{AB}| \geq (|q_A| - |q_B|) - 1$" condition corresponds to the case in which there are enough number of replicated data items requested by q that can be utilized to achieve even distribution of q among $\mathcal{D}_A$ and $\mathcal{D}_B$. The "otherwise" condition corresponds to the case for which even distribution of q among the disks is not possible. In the former case, the replicated data items requested by q will be retrieved from $\mathcal{D}_A$ and $\mathcal{D}_B$ in an appropriate manner to attain even distribution, whereas in the latter case, all of the replicated data items requested by q will be retrieved from $\mathcal{D}_B$ to minimize the cost of query q. Hence, for a two-way replicated declustering $R_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$, the cost $r(q)$ of q can be computed with the following closed-form expression:

$$r(q) = \begin{cases} \lceil |q|/2 \rceil & \text{if } |q_{AB}| \geq (||q_A| - |q_B||) - 1 \\ max(t_A(q), t_B(q)) & \text{otherwise.} \end{cases} \tag{2.10}$$

The simple closed-form expressions given in Equations 2.8, 2.9, and 2.10 for computing $r(q)$ enable us to avoid constructing the optimal schedules for the queries throughout the iterations of the two-way replicated declustering algorithm. That is, $r(q)$ in Equation 2.10 gives the cost of query q that can be attained by an optimal schedule for q, without constructing $S_{opt}(q)$ through costly network-flow based algorithms.

It is clear that optimizing the cost function given below at each two-way replicated declustering step will optimize the "goodness" criteria explained at Section 2.4.1:

$$cost(R_2) = \sum_{q \in Q} f(q)(r(q) - \lceil |q|/2 \rceil). \tag{2.11}$$

Our overall two-way replicated declustering algorithm works as a sequence of two-way refinement passes performed over all data items. In each pass, we start with computing the initial operation gains of all data items. Then, we iteratively perform the following computations: find the data item and the operation that produces the highest reduction in the cost; perform that operation; update gain values of neighboring data items; lock the selected data item to further processing

to prevent thrashing.

We perform these computations until there are no remaining data items to process. We restore the declustering to the state where the best reduction is obtained during the pass and we start a new pass over the data items if the obtained improvement in the current pass is above a threshold or if the number of passes performed is below some predetermined number. Once we obtain a two-way declustering, we can recursively apply our two-way declustering algorithm on each of these declusters to obtain any number of declusters. A running example demonstrating move, replication and unreplication gain updates can be found in Section 2.5.

All operations are kept in priority queues keyed according to their gain values. The priority queues are implemented as binary heaps. For a two-way declustering, we maintain six heaps: Two heaps for storing the move operations of data items from part A to B and from part B to A, two heaps for storing the replication operations of data items from part A to B and from part B to A, and two heaps for storing the unreplication operations of replicated data items from part A and from part B.

In our two-way replicated declustering algorithm, we start with calculating the initial move, replication and unreplication gains of all data items (Section 2.6, Algorithm 2) . After initializing the gains, we retrieve the highest gains and the associated data items for each operation type and by comparing these gains we select the best operation to perform. If there are any possible unreplication operations which do not increase the total cost of the system (i.e., with zero unreplication gain), those unreplication operations are performed first. After we finish possible unreplications, we compare the gains to be obtained by move and replication operations. If the gains are the same, we prefer to perform move operations. Recall that each data item is eligible for two types of operations and thus has two related gain values. So, after deciding on the best operation to perform, we remove the data item from the two related heaps by extractMax and delete operations.

After performing an operation (move, replication or unreplication) on a data item d^* , we may need to update the gains of operations related with the data items that are neighbor to d^* (Section 2.6, Algorithms 3, 4, and 5). For any data item d, we have $g_r(d) \geq g_m(d)$, hence, in a pass, the number of replication operations tend to outweigh the number of move operations. A similar problem had been observed when replication was used for clustering purposes in the VLSI literature and one of the solutions proposed was the gradient methodology [25]. We adopt this methodology by permitting solely move and unreplication operations until the improvement obtained drops below a certain threshold and only after that we perform replication operations.

2.4.1.2 Query splitting

At the end of a two-way replicated declustering $R_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$ of a dataset and query set pair $\{\mathcal{D}, Q\}$, we split the queries of Q among the obtained two sub-datasets as evenly as possible so that split queries correctly represent the optimizations performed during that two-way replicated declustering step. That is, an R_2 is decoded as splitting each query $q \in Q$ into two sub-queries

$$q' \subseteq q \cap \mathcal{D}_A \text{ and } q'' \subseteq q \cap \mathcal{D}_B, \tag{2.12}$$

such that the difference $||q'| - |q''||$ is minimized. The split queries q' and q'' are added to sub-query sets Q_A and Q_B, respectively so that further two-way declustering operations can be recursively performed on $\{\mathcal{D}_A, Q_A\}$ and $\{\mathcal{D}_B, Q_B\}$ pairs.

Recall that the optimizations performed during a two-way replicated declustering assume that queries will have optimal schedules with regard to that of two-way replicated declustering, and even splitting of queries ensures that. Also recall that constructing the optimal schedule of a query q in a replicated declustering system requires network-flow-based algorithms. However, for two-way replicated declustering this feat can be achieved by utilizing the item distribution $dist(q)$ of q and the value of $r(q)$, which can be computed via the closed form definitions

given in Equations 2.7–2.10. We know that in an optimal splitting according to the optimal schedule, the size of q' should be $|q'| = t_A(q)$ and the size of q'' should be $|q''| = t_B(q)$.

Consider the three-way partition of query q into q_A, q_B, and q_{AB} (according to Equation 2.8) induced by the two-way replicated declustering. It is clear that data items in q_A will go into q' and data items in q_B will go into q'', so all that remains is to decide on the splitting of the data items in q_{AB} according to an optimal scheduling. Let us call the replicated data items that will go into q' as q'_{AB} and the replicated data items that will go into q'' as q''_{AB}. That is,

$$q' = q_A \cup q'_{AB} \text{ and } q'' = q_B \cup q''_{AB}, \tag{2.13}$$

Since we want to enforce a splitting such that $|q'| = t_A(q)$ and $|q''| = t_B(q)$, we can say that

$$|q'_{AB}| = t_A(q) - |q_A| \text{ and } |q''_{AB}| = t_B(q) - |q_B| = |q_{AB}| - |q'_{AB}|. \tag{2.14}$$

Any splitting of the data items in q_{AB} that respects the size constraints given in Equation 2.14 satisfies the optimality condition. In our studies we assign the first $t_A(q) - |q_A|$ items of q_{AB} to q' and the remaining items of q_{AB} to q''. Other assignment schemes can be explored for better performance results.

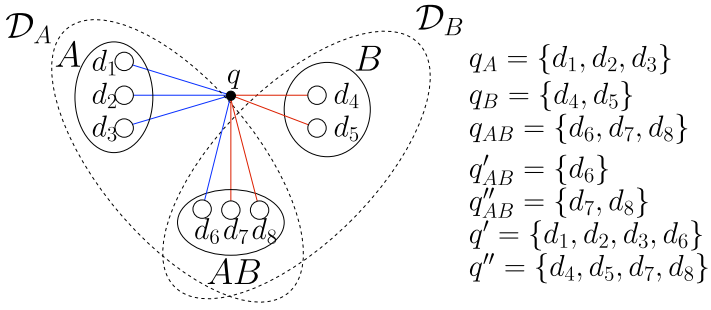


Figure 2.2: Splitting of a query q according to a two-way replicated declustering $R_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$.

A sample splitting of a query q with eight data items is given in Fig. 2.2.

According to Equation 2.9, for q, $t_A(q) = 4$ and $t_B(q) = 4$, hence $|q'| = 4$ and $|q''| = 4$. Since, for q, $|q_A| = 3$, $|q_B| = 2$, $|q_{AB}| = 3$, by Equations 2.13 and 2.14, we can say that $|q'_{AB}| = t_A(q) - |q_A| = 1$ and $|q''_{AB}| = t_B(q) - |q_B| = 2$. Any splitting of q_{AB} according to these size constraints satisfies the optimality condition, and according to our assignment scheme $q'_{AB} = \{d_6\}$ and $q''_{AB} = \{d_7, d_8\}$. Hence, $q' = q_A \cup q'_{AB} = \{d_1, d_2, d_3, d_6\}$ and $q'' = q_B \cup q''_{AB} = \{d_4, d_5, d_7, d_8\}$.

2.4.2 Multi-way replicated refinement

Our multi-way replicated refinement scheme starts with the K-way replicated declustering of the dataset $\mathcal{D}$, say $R_K = \{\mathcal{D}_1, \ldots, \mathcal{D}_K\}$, generated by the recursive replicated declustering scheme described in Section 2.4.1. We iteratively improve R_K by multi-way refinement operations K-way *move* and K-way *replication*. In order to perform these operations we maintain the following gain values for each data item d:

- K-way move gain ($g_m(d, k)$): the reduction to be observed in the overall query processing cost, if d is moved to disk k,
- K-way replication gain ($g_r(d, k)$): the reduction to be observed in the overall query processing cost, if d is replicated in disk k.

If we were to maintain the above gain values for all data items, we would need approximately $2 \times (K-1)$ gain values for each data item, because a data item can be moved or replicated from its current source disk(s) to any of the disks that does not already store it. Instead of this expensive schema, we adapt an efficient greedy approach that was proposed for unreplicated declustering in [3] to support multi-way refinement and we develop a multi-way refinement heuristic suitable for replicated declustering. Our heuristic can perform multi-way move and replication operations. The approach in [3] was based on the observation that a move operation can be viewed as a two-stage process, where in the first stage the data item d^* to be moved is assumed to leave the source disk and in the second stage d^* arrives at the destination disk. The first stage represents the decrease in the load of the source disk due to the relief in processing of the

queries related with d^*, resulting with a decrease in the cost. The second stage represents the increase in the load of the destination disk due to the excess in processing of the queries related with d^*, resulting with an increase in the cost. Here we extend this efficient greedy approach to support both multi-way move and replication selection operations. Our adapted schema requires maintenance of only a single gain value (virtual leave gain) for each data item d.

Virtual leave gain $vg(d)$ indicates the number of queries requesting d such that the disk(s) that d resides in serve(s) more than optimal number of data items for these queries. That is, the virtual leave gain of a data item d that resides on disk D_s is:

$$vg(d) = \sum_{q \in Q_+(d,s)} f(q), \text{where} \tag{2.15}$$

$$Q_+(d,s) = \{q \in Q : d \in q \wedge t_s(q) > r_{opt}(q)\} \tag{2.16}$$

That is, each query q that requests data item d contributes $f(q)$ to $vg(d)$, if the number of data items in q that are retrieved from disk D_s is greater than the optimal response time $r_{opt}(q)$ of q. This means that it is possible to improve the distribution of query q through moving or replicating data item d to an appropriate destination disk D_z. Thus, virtual leave gain is an upper bound on the actual move or replication gain. We should note here that our definition of virtual leave gain is different from that of [3] in order to support correct computation of multi-way move and multi-way replication operations. A running example demonstrating virtual leave gain updates can be found in Section 2.5.2.

Our overall K-way replicated declustering refinement algorithm works as a sequence of multi-way passes performed over all data items. Before starting the multi-way refinement passes, as a preprocessing step, we compute the optimal schedules for all queries once and maintain these schedules in a data structure called *OptSched*. The process of initial optimal schedule calculation is performed using network-flow based algorithms [12]. *OptSched* is composed of $|Q|$ arrays, where the ith array is of size $|q_i|$ and stores from which disks the data items of q_i

are answered in the optimal scheduling. *OptSched* is kept both to identify bottleneck disks for queries and also to report the actual aggregate parallel response time of the replicated declustering produced by the recursive declustering phase. A bottleneck disk for a query q is the disk from which q requests the maximum number of data items (and hence determines response time $r(q)$).

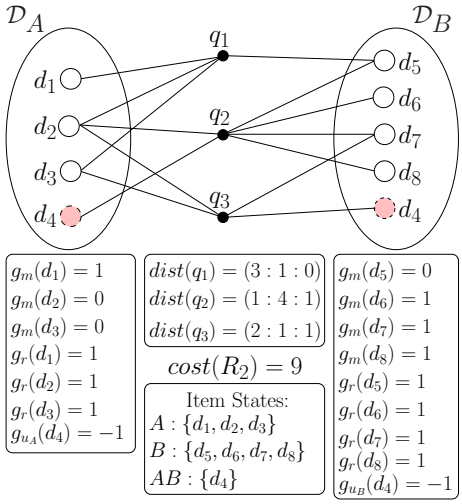
In a multi-way refinement pass, we start with computing the virtual leave gains of all data items (Section 2.5.2, Algorithm 6). At each iteration of a pass, a data item d^* with the highest virtual leave gain is selected. The $K-1$ move and $K-1$ replication gains associated with d^* are computed (Section 2.5.2, Algorithm 7), the best operation associated with d^* is selected and performed if it has a positive actual gain and if it obeys the given capacity constraints, and then the virtual leave gain values of the neighboring data items of d^* are updated (Section 2.5.2, Algorithm 8). Also the optimal schedules of each query that requests d^* is considered for update in constant time by investigating possible changes in the bottleneck disk of that query. We perform these passes until the obtained improvement is below a certain threshold or we reach a predetermined number of passes.

2.5 Examples for two-way and multi-way refinement schemes

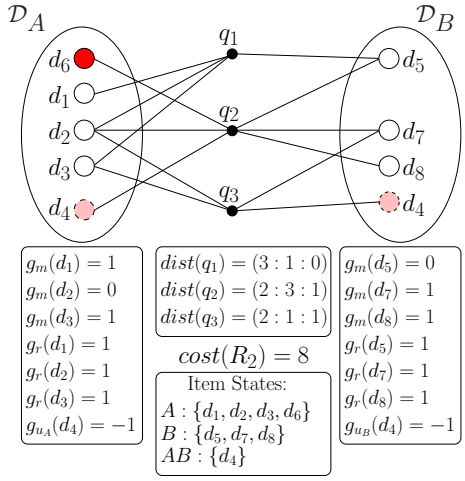
In this section, the running of our two-way and multi-way refinement schemes are demonstrated.

2.5.1 Two-way refinement scheme example

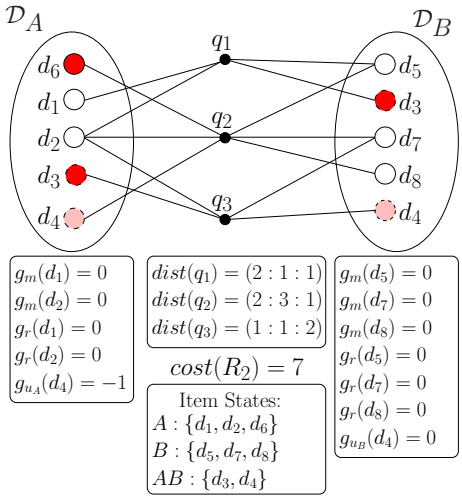
In the sample query set and two-way declustering given in Fig. 2.3(a), the query distribution for q_3 is $dist(q_3) = (2 : 1 : 1)$, since q_3 requests d_2 and d_3 which are in part A, d_7 which is in part B, and d_4 which is replicated and hence in part AB. Likewise, in Fig. 2.3(a), the retrieval times of query q_3 from disks $\mathcal{D}_A$ and $\mathcal{D}_B$



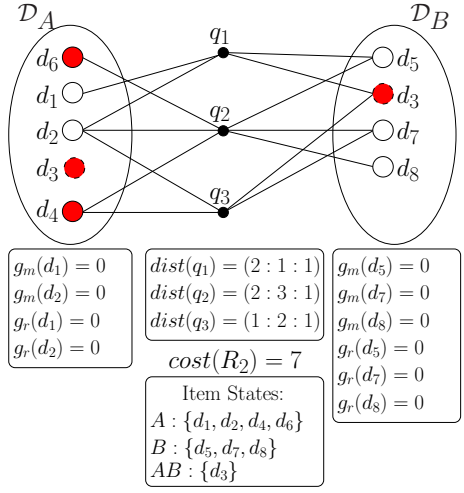
(a) Initial declustering



(b) After the move of d_6



(c) After the replication of d_3



(d) After the unreplication of d_4

Figure 2.3: Gain values of operations defined over the data items and item distributions of queries for a sample two-way declustering with 8 data items and 3 queries.

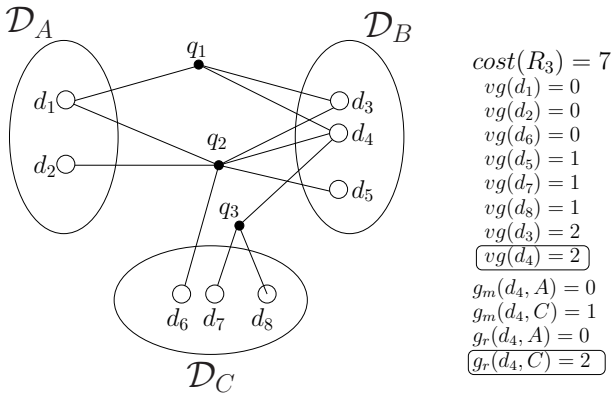
are $t_A(q_3) = 2$ and $t_B(q_3) = 2$, respectively. The cost of the two-way replicated declustering R_2 in Fig. 2.3(a) is $cost(R_2) = r(q_1) + r(q_2) + r(q_3) = 3 + 4 + 2 = 9$.

In Fig. 2.3, the effect of move (Fig. 2.3(b)), replication (Fig. 2.3(c)) and unreplication (Fig. 2.3(d)) operations on the gains of data items, the item distribution of queries and the total cost of the replicated declustering are exampled over a sample initial two-way declustering given in Fig. 2.3(a). In Fig. 2.3(a), the move gain of d_6 is $g_m(d_6) = 1$ and after the move of d_6 from $\mathcal{D}_B$ to $\mathcal{D}_A$, the cost of the declustering reduces by one as shown in Fig. 2.3(b). Similarly, in Fig. 2.3(b), the replication gain of d_3 is $g_r(d_3) = 1$ and after the replication of d_3 from $\mathcal{D}_A$ to $\mathcal{D}_B$, the cost of the declustering reduces by one more as shown in Fig. 2.3(c). In Fig. 2.3(c), the unreplication-from-B gain of d_4 is $g_{u_B}(d_4) = 0$ and after the unreplication of d_4 from $\mathcal{D}_B$, the cost of the declustering remains the same as shown in Fig. 2.3(d).

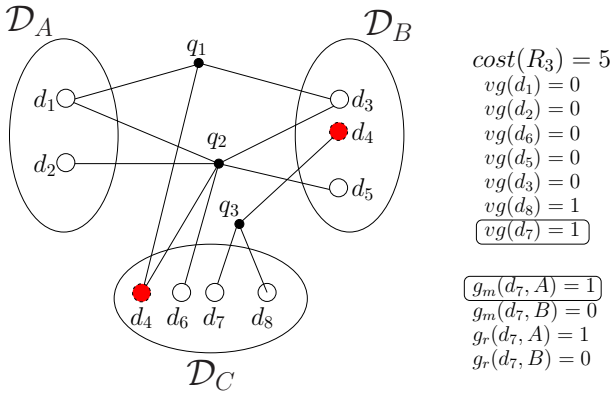
Note that the cost of the declustering reduces by one after the move in compliance with the move gain of d_6. Fig. 2.3(c) shows the changes in data item gains and query distributions after the replication of d_6 from $\mathcal{D}_B$ to $\mathcal{D}_A$. Note that the cost of the declustering reduces by one after the move in compliance with the move gain of d_6.

2.5.2 Multi-way refinement scheme example

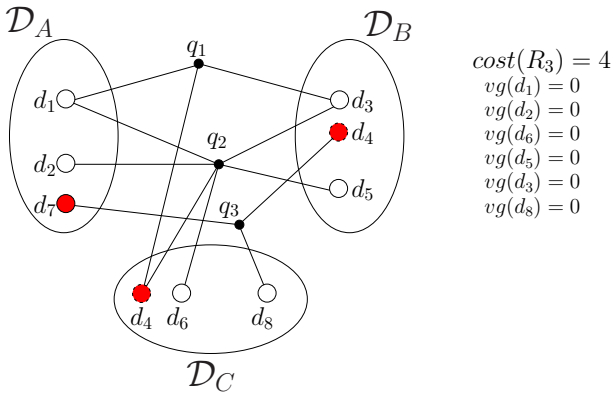
We present a running example of how our multi-way refinement scheme works over the sample query set and three-way declustering given in Fig. 2.4(a), In Fig. 2.4, the virtual leave gains of data items and the actual move gains of the data item with the highest virtual leave gain are displayed. In Fig. 2.4(a), d_3 and d_4 both have a virtual leave gain of two, which is the highest virtual leave gain. We select d_4 and compute its actual move and replication gains. Replicating d_4 to $\mathcal{D}_C$ provides the highest actual gain ($g_r(d_4, C) = 2$). Fig. 2.4(b) displays the query distribution and the changes in the virtual leave gains after the replication. Note that the cost of the declustering reduces by two after the replication in compliance with the replication gain of d_4 to $\mathcal{D}_C$. In Fig. 2.4(b), d_7 and d_8 both



(a) A three-way declustering



(b) After the replication of d_4 to $\mathcal{D}_C$



(c) After moving d_7 to $\mathcal{D}_A$

Figure 2.4: Virtual leave gain values of data items and actual gain of the highest virtual leave gained data item for a sample three-way declustering with 8 data items and 3 queries.

have a virtual leave gain of one, which is the highest virtual leave gain. We select d_7 and compute its actual move and replication gains. Moving d_7 to $\mathcal{D}_A$ provides the highest actual gain ($g_m(d_7, A) = 1$). Fig. 2.4(c) displays the query distribution and the changes in the virtual leave gains after the move operation.

2.6 Details of Selective Replicated Assignment Algorithms

In this section, we present the details of the algorithms used in our Selective Replicated Assignment (SRA) scheme.

2.6.1 Recursive replicated declustering algorithms

Algorithm 1: DeltaCalculation(Query q, Part p).

Require: $q \in \mathcal{Q}$, $p \in \{A, B\}$
1: **if** $|t_A(q) - t_B(q)| < t_{AB}(q)$ **then**
2: **if** $|q|\%2 = 0$ **then**
3: **return** $\Delta \leftarrow 0$
4: **else**
5: **return** $\Delta \leftarrow 1$
6: **else**
7: **if** $t_p(q) < |q| - t_{AB}(q) - t_p(q)$ **then**
8: **return** $\Delta \leftarrow 2 \times t_p(q) + 2 \times t_{AB}(q) - |q|$
9: **else**
10: **return** $\Delta \leftarrow 2 \times t_p(q) - |q|$

In SRA algorithms, delta (Δ) of a query relative to a part, which is a notion that relates to the difference between the cost of a query from its optimal distribution cost, is frequently used. Part information decides on the sign of the Δ value. Δ of a query q is positive for the part that q requires more data items, whereas it is negative for the part it requires less data items. Δ calculation is given in Algorithm 1. For a given query q, if the query item distribution is such that all replicas will not be requested from the same disk (i.e., $|t_A(q) - t_B(q)| < t_{AB}(q)$), then it is possible to distribute the query as evenly as possible by utilizing the

replicas, hence Δ is either zero or one depending on the query size. This case is covered in lines 1–5. In lines 6–10, the case where all replicas will be requested from one disk is handled. In this case the replicated data items are thought as fixed to that part, and Δ is calculated accordingly. For example, for the sample dataset given in Fig. 2.3(a), Δ of q_2 for part A is -2 and for part B is 2.

Algorithm 2: Initialize gains.

Require: $(\mathcal{D}, \mathcal{Q})$, $\Pi_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$
1: **for each** query $q \in \mathcal{Q}$ **do**
2: $t_A(q) \leftarrow t_B(q) \leftarrow t_{AB}(q) \leftarrow 0$
3: **for each** data item $d \in q$ **do**
4: $k \leftarrow State(d)$, $t_k(q) \leftarrow t_k(q) + 1$
5: **for each** data item $d \in \mathcal{D}$ **do**
6: **if** $State(d) \neq AB$ **then**
7: $g_m(d) \leftarrow g_r(d) \leftarrow 0$, $k \leftarrow State(d)$
8: **for each** query q that contains d **do**
9: $\Delta \leftarrow DeltaCalculation(q, k)$
10: **if** $\Delta \geq 2$ **then**
11: $g_m(d) \leftarrow g_m(d) + f(q)$, $g_r(d) \leftarrow g_r(d) + f(q)$
12: **else if** $(\Delta = 0) \wedge (2(t_k(q) + t_{AB}(q)) = |q|)$ **then**
13: $g_m(d) \leftarrow g_m(d) - f(q)$
14: **else if** $\Delta \leq -1$ **then**
15: $g_m(d) \leftarrow g_m(d) - f(q)$
16: **else if** $State(d) = AB$ **then**
17: $g_{u_A}(d) \leftarrow g_{u_B}(d) \leftarrow 0$
18: **for each** query q that contains d **do**
19: **if** $t_A(q) \geq t_B(q) + t_{AB}(q)$ **then**
20: $g_{u_B}(d) \leftarrow g_{u_B}(d) - f(q)$
21: **else if** $t_B(q) \geq t_A(q) + t_{AB}(q)$ **then**
22: $g_{u_A}(d) \leftarrow g_{u_A}(d) - f(q)$

Algorithm 2 is used in calculating the initial move, replication and unreplication gains of all data items. After calculating query item distributions of all queries (lines 1–4), we calculate the move and replication gains of unreplicated data items (lines 6–15). In lines 8–15, we update the gains of each data item via investigating each query q that requests it. Lines 10–11 handles the case where Δ of the investigated query is greater than or equal to 2. In this case, the move and replication gains of data item d increases by $f(q)$, since moving or replicating d to the other part would reduce the cost of that query by $f(q)$. Lines 12–13 handles the case where Δ equals to 0 and all replicated data items are retrieved from the same disk in which d resides. In this case, the move gain of d is reduced by $f(q)$ since moving d would increase the cost of investigated query by $f(q)$,

however, the replication gain of d does not change since for that particular query, it is possible to use one of the replicas in the other part. Lines 14–15 handles the case where $\Delta \geq -1$. This means that for query q, the number of data items requested from the disk in which d resides is very small compared to the number of data items requested from the other disk, hence the gain of moving data item d from its current part to the other part is reduced by $f(q)$ (line 15). In lines 16–22 of Algorithm 2, unreplication gains of replicated data items are calculated. We update unreplication gains by investigating each query. If all replicated data items are retrieved from disk $\mathcal{D}_B$, we reduce the unreplication gain of the replica of d from $\mathcal{D}_B$ (lines 19–20), whereas if all replicated data items are retrieved from disk $\mathcal{D}_A$, we reduce the unreplication gain of the replica of d from $\mathcal{D}_A$ (lines 21–22).

Algorithm 3 is used to update the gains of necessary data items after moving data item d^* from part A to part B. In lines 3–28, the algorithm handles unreplicated data items. In lines 4–16, the data items that are initially at the same part with d^* are handled. For those data items, if $\Delta = 3$ (lines 5–6), reduction of move and replication gains by $f(q)$ is sufficient. If $\Delta = 2$ (lines 7–12), again reduction of replication gain by $f(q)$ suffices, but the move gain requires special care. If there is at least one replicated data item requested by q, then reduction of move gain by $f(q)$ is sufficient (lines 9–10), otherwise reduction of move gain by $2f(q)$ is necessary since Δ will be zero after the move (lines 11–12). Lines 13–14 handles the case where $\Delta = 1$ and all replicated data items are retrieved from $\mathcal{D}_A$. In this case, the move gain is reduced by $f(q)$ since after the move of d^*, Δ becomes -1 and the move of d would increase the cost of investigated query by $f(q)$, however, the replication gain does not change since for that particular query the replication of d from A to B would not change Δ. Lines 15–16 handles the case where $\Delta = 0$ and only one replicated data item is retrieved from B. In this case, the move gain is reduced by $f(q)$ since after the move of d^*, Δ is still 0 (since we can use the replica in part A) but further moves from A to B (e.g., moving d) would increase the cost of query q by $f(q)$, however, the replication gain does not change.

In Algorithm 3, lines 17–28 updates the gains of data items that are initially

Algorithm 3: Update gains after a move to $\mathcal{D}_B$.

Require: $(\mathcal{D}, \mathcal{Q})$, $\Pi_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$, $State(d^*) = A$

1: **for each** query $q \in \mathcal{Q}$ that contains d^* **do**
2: $\Delta \leftarrow$ DeltaCalculation(q, A)
3: **for each** unreplicated, unlocked data item $d \in q$ **do**
4: **if** $d \in \mathcal{D}_A$ **then**
5: **if** $\Delta = 3$ **then**
6: $g_m(d) \leftarrow g_m(d) - f(q)$, $g_r(d) \leftarrow g_r(d) - f(q)$
7: **else if** $\Delta = 2$ **then**
8: $g_r(d) \leftarrow g_r(d) - f(q)$
9: **if** $t_{AB}(q) \geq 1$ **then**
10: $g_m(d) \leftarrow g_m(d) - f(q)$
11: **else**
12: $g_m(d) \leftarrow g_m(d) - 2f(q)$
13: **else if** $\Delta = 1 \wedge (t_A(q) + t_{AB}(q) = t_B(q) + 1)$ **then**
14: $g_m(d) \leftarrow g_m(d) - f(q)$
15: **else if** $\Delta = 0 \wedge |q| = 2(t_B(q) + 1)$ **then**
16: $g_m(d) \leftarrow g_m(d) - f(q)$
17: **else if** $d \in \mathcal{D}_B$ **then**
18: **if** $\Delta = 1 \wedge (t_A(q) + t_{AB}(q) = t_B(q) + 1)$ **then**
19: $g_m(d) \leftarrow g_m(d) + f(q)$
20: **else if** $\Delta = 0$ **then**
21: **if** $t_{AB}(q) = 0$ **then**
22: $g_m(d) \leftarrow g_m(d) + 2f(q)$, $g_r(d) \leftarrow g_r(d) + f(q)$
23: **else if** $|t_A(q) - t_B(q)| = t_{AB}(q)$ **then**
24: $g_m(d) \leftarrow g_m(d) + f(q)$
25: **if** $t_B(q) - t_A(q) = t_{AB}(q)$ **then**
26: $g_r(d) \leftarrow g_r(d) + f(q)$
27: **else if** $\Delta = -1$ **then**
28: $g_m(d) \leftarrow g_m(d) + f(q)$, $g_r(d) \leftarrow g_r(d) + f(q)$
29: **for each** replicated, unlocked data item $d \in q$ **do**
30: **if** $t_B(q) + t_{AB}(q) + 2 > t_A(q) \geq t_B(q) + t_{AB}(q)$ **then**
31: $g_{u_B}(d) \leftarrow g_{u_B}(d) + f(q)$
32: **if** $t_A(q) - t_B(q) = 1$ **then**
33: $g_{u_A}(d) \leftarrow g_{u_A}(d) - f(q)$
34: **if** $|t_A(q) - t_B(q)| < t_{AB}(q) \leq |t_A(q) - t_B(q) - 2|$ **then**
35: $g_{u_A}(d) \leftarrow g_{u_A}(d) - f(q)$
36: $t_A(q) \leftarrow t_A(q) - 1$, $t_B(q) \leftarrow t_B(q) + 1$
37: $State(d^*) \leftarrow B$, $Locked(d^*) \leftarrow 1$

in part B. In lines 18–19, the case of $\Delta = 1$ and the replicas of replicated data items of q are all retrieved from disk $\mathcal{D}_B$ is examined. In this case, after the move of d^* to $\mathcal{D}_B$, Δ becomes -1 and moves from B to A become viable (they will not increase the cost of q), this explains the increase in the move gain of d. Lines 21–22 covers the case where $\Delta = 0$ and none of the data items requested by the query is replicated. In this case, since there is no replication, after the move of d^*, Δ becomes -2 and thus the move gains of data items in disk $\mathcal{D}_B$ that are neighbor to d^* via q increase by $2f(q)$ (move gain changes from -$f(q)$ to $f(q)$). However, the replication gain only increases by $f(q)$ (since it changes from 0 to $f(q)$). Lines 23–26 indicate the case where $\Delta = 0$ and the replicas of replicated data items of q are all retrieved from the same disk. In this case, after the move of d^* the move gain of d increases by $f(q)$. Specifically, if the replicated data items of q are all retrieved from disk $\mathcal{D}_A$ (lines 25–26) then the replication gain increases as well, since after the move Δ will be -1 and replication of d from B to A will reduce the cost of the query. The case where Δ of the query is equal to -1 is examined in lines 27–28. In this case, after the move of d^*, Δ will be -2, hence both replication and move from $\mathcal{D}_B$ to $\mathcal{D}_A$ will improve the cost of the query.

In Algorithm 3, lines 29–36 updates the unreplication gains of replicated data items that are neighbors to d^*. For a specific query q that requests d^*, if the replicated data items of q are all retrieved from disk $\mathcal{D}_B$ and the number of these items is greater than $t_A(q) - t_B(q) - 2$, then unreplication of d from $\mathcal{D}_B$ becomes viable for q (lines 30–33). Specifically, if $t_A(q) - t_B(q) = 1$ then unreplication of d from $\mathcal{D}_A$ increases the cost of q by 1 (lines 32–33), hence unreplication gain of d from $\mathcal{D}_A$ is reduced by $f(q)$. Lines 34–35 handles the case where after the move of d^* to B, the replicas of replicated data items of q will all be retrieved from disk $\mathcal{D}_A$. In this case, unreplication of d from $\mathcal{D}_A$ will increase the cost of q by 1.

Algorithm 4 updates the gains of necessary data items after replicating a data item d^* from $\mathcal{D}_A$ to $\mathcal{D}_B$. In lines 3–11, the algorithm updates the gains of unreplicated neighbors of d^*. In lines 4–6, the gains of data items that are stored in $\mathcal{D}_A$ are updated. If $\Delta = 3$ or $\Delta = 2$, after the replication of d^*, query q will start retrieving d^* from $\mathcal{D}_B$ and Δ will reduce to 0 or 1, hence the move and replication gains of data items from $\mathcal{D}_A$ to $\mathcal{D}_B$ will reduce. In lines 7–11, the gains of data

Algorithm 4: Update gains after a replication to $\mathcal{D}_B$.

Require: $(\mathcal{D}, \mathcal{Q})$, $\Pi_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$, $State(d^*) = A$
1: **for each** query $q \in \mathcal{Q}$ that contains d^* **do**
2: $\Delta \leftarrow$ DeltaCalculation(q, A)
3: **for each** unreplicated, unlocked data item $d \in q$ **do**
4: **if** $d \in \mathcal{D}_A$ **then**
5: **if** $\Delta = 3 \vee \Delta = 2$ **then**
6: $g_m(d) \leftarrow g_m(d) - f(q)$, $g_r(d) \leftarrow g_r(d) - f(q)$
7: **else if** $d \in \mathcal{D}_B$ **then**
8: **if** $\Delta = 1 \wedge (t_A(q) = t_{AB}(q) + t_B(q) + 1)$ **then**
9: $g_m(d) \leftarrow g_m(d) + f(q)$
10: **else if** $\Delta = 0 \wedge (t_A(q) = t_{AB}(q) + t_B(q))$ **then**
11: $g_m(d) \leftarrow g_m(d) + f(q)$
12: **for each** replicated, unlocked data item $d \in q$ **do**
13: **if** $t_B(q) + t_{AB}(q) + 2 > t_A(q) \geq t_B(q) + t_{AB}(q)$ **then**
14: $g_{u_B}(d) \leftarrow g_{u_B}(d) + f(q)$
15: $t_A(q) \leftarrow t_A(q) - 1$, $t_{AB}(q) \leftarrow t_{AB}(q) + 1$
16: $State(d^*) \leftarrow AB$, $Locked(d^*) \leftarrow 1$

items that are stored in $\mathcal{D}_B$ are updated. Lines 8–9 handle the case where $\Delta = 1$ and the replicated data items of q are all retrieved from $\mathcal{D}_B$. Lines 10–11 handle the case where $\Delta = 0$ and the replicated data items of q are all retrieved from $\mathcal{D}_B$. In these cases, prior to the replication of d^*, moves from $\mathcal{D}_B$ to $\mathcal{D}_A$ would increase the cost of q, but after the replication, such moves will not affect the cost of q, hence the move gain of d is increased. In lines 12–14, the unreplication gains of unlocked and replicated data items are updated. If the replicated data items of q will all be retrieved from disk $\mathcal{D}_B$ and $-1 \geq \Delta \geq 2$, then after the replication of d^* to $\mathcal{D}_B$, unreplicating a replicated data item from $\mathcal{D}_B$ becomes viable (lines 13–14).

Algorithm 5 updates the gains of necessary data items after unreplicating a replica of d^* from $\mathcal{D}_A$. In lines 3–13, the algorithm updates the gains of unreplicated data items. In lines 4–8, for each query q that requests d^*, we investigate whether the unreplication of d^* from $\mathcal{D}_A$ removes the flexibility in moving d^*'s neighboring data items from A to B. If this is the case so, we reduce the move gains of such data items by $f(q)$. Lines 9–13 cover the case where the unreplication increases the cost of the query by increasing the number of data items retrieved from $\mathcal{D}_B$. In this case, the move and replication gains of unreplicated data items in part B are increased by $f(q)$. In lines 14–16, the unreplication

Algorithm 5: Update gains after unreplication from $\mathcal{D}_A$.

Require: $(\mathcal{D}, \mathcal{Q})$, $\Pi_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$, $State(d^*) = AB$
1: **for each** query $q \in \mathcal{Q}$ that contains d^* **do**
2: $\Delta \leftarrow$ DeltaCalculation(q, A)
3: **for each** unreplicated, unlocked data item $d \in q$ **do**
4: **if** $d \in \mathcal{D}_A$ **then**
5: **if** $\Delta = 0 \wedge 2 \leq t_A(q) + t_{AB}(q) - t_B(q) \leq 3$ **then**
6: $g_m(d) \leftarrow g_m(d) - f(q)$
7: **else if** $\Delta = 1 \wedge t_A(q) + t_{AB}(q) - t_B(q) = 1$ **then**
8: $g_m(d) \leftarrow g_m(d) - f(q)$
9: **else if** $d \in \mathcal{D}_B$ **then**
10: **if** $\Delta = 0 \wedge (t_A(q) + t_{AB}(q) = t_B(q))$ **then**
11: $g_m(d) \leftarrow g_m(d) + f(q)$, $g_r(d) \leftarrow g_r(d) + f(q)$
12: **else if** $\Delta = -1$ **then**
13: $g_m(d) \leftarrow g_m(d) + f(q)$, $g_r(d) \leftarrow g_r(d) + f(q)$
14: **for each** replicated, unlocked data item $d \in q$ **do**
15: **if** $|t_B(q) - t_A(q)| < t_{AB}(q) \leq t_B(q) - t_A(q) + 2$ **then**
16: $g_{u_A}(d) \leftarrow g_{u_A}(d) - f(q)$
17: $t_B(q) \leftarrow t_B(q) + 1$, $t_{AB}(q) \leftarrow t_{AB}(q) - 1$
18: $State(d^*) \leftarrow B$, $Locked(d^*) \leftarrow 1$

gains of unlocked and replicated data items are updated in the case where after the unreplication of d^* the replicas in $\mathcal{D}_A$ are being used. In this case, the unreplication-from-A gains of replicated data items decreases.

2.6.2 Multi-Way Replicated Declustering Algorithms

Algorithm 6: Initialize virtual leave gains.

Require: $(\mathcal{D}, \mathcal{Q})$, $\Pi_K = \{\mathcal{D}_1, \ldots, \mathcal{D}_K\}$
1: **for each** query $q \in \mathcal{Q}$ **do**
2: $r_{opt}(q) \leftarrow \lceil |q|/K \rceil$
3: $r(q) \leftarrow max_{1 \leq k \leq K}\{t_k(q)\}$
4: **for each** data item $d \in \mathcal{D}$ **do**
5: $vg(d) \leftarrow 0$
6: **for each** query $q \in \mathcal{Q}$ that contains d **do**
7: $k \leftarrow getRequestedDisk(q, d)$
8: **if** $t_k(q) > r_{opt}(q)$ **then**
9: $vg(d) \leftarrow vg(d) + f(q)$

Algorithm 6 describes how we initialize the virtual leave gains of data items. In line 2 we set the optimal retrieval time of each query and in line 3 we calculate the maximum number of documents retrieved for query q from each disk. The

for loop between lines 4–9 computes the virtual leave gain value ($vg(d)$) for all data items. We first initialize the virtual leave gain of a data item d to 0 (line 5) and then, for each query q that requests d, we determine from which disk q requests d (line 7). If the number of data items requested by q from that disk is higher than $r_{opt}(q)$, we increase the virtual leave gain of d by $f(q)$ (line 9). Note that an $O(|q|)$-time $getRequestedDisk(q, d)$ operation is required to find which replica of d is used for answering q by examining the *OptSched* data structure.

Algorithm 7: Compute actual gains.

Require: $(\mathcal{D}, \mathcal{Q})$, $\Pi_K = \{\mathcal{D}_1, \ldots, \mathcal{D}_K\}, d^* \in \mathcal{D}_s$
1: **for each** disk $\mathcal{D}_k$, $k \in \{1 \ldots K\}, k \neq s$ **do**
2: $g_m(d^*, k) \leftarrow 0$, $g_r(d^*, k) \leftarrow 0$
3: **if** d^* is replicated to disk $\mathcal{D}_k$ **then**
4: continue
5: **for each** query q that contains d^* **do**
6: **if** d^* is not replicated **then**
7: **if** $t_s(q) = r(q) \wedge r(q) > r_{opt}(q)$ **then**
8: **if** $t_k(q) < r(q) - 1$ **then**
9: $g_m(d^*, k) \leftarrow g_m(d^*, k) + f(q)$
10: $g_r(d^*, k) \leftarrow g_r(d^*, k) + f(q)$
11: **else if** $t_k(q) \geq r(q)$ **then**
12: $g_m(d^*, k) \leftarrow g_m(d^*, k) - f(q)$
13: **else**
14: $m \leftarrow getRequestedDisk(q, d^*)$
15: **if** $t_m(q) = r(q) \wedge r(q) > r_{opt}(q)$ **then**
16: **if** $t_k(q) < r(q) - 1$ **then**
17: $g_r(d^*, k) \leftarrow g_r(d^*, k) + f(q)$

Algorithm 7 computes the actual gains obtained by moving or replicating a data item $d^* \in \mathcal{D}_s$ to any other disk. For each candidate disk $\mathcal{D}_k$, the algorithm initially sets the actual move gain ($g_m(d^*, k)$) and the actual replication game ($g_r(d^*, k)$) to 0 (line 2). If d^* is already replicated in $\mathcal{D}_k$ (lines 3–4) we do not need to compute any gains, otherwise, we check each query q requesting d^*. We first check whether d^* is replicated (line 6). If it is not and if the data items requested by q are not optimally distributed and disk $\mathcal{D}_s$ is among the bottleneck disks of q (line 7), we increase the actual move and replication gain of d^* to $\mathcal{D}_k$ (lines 9–10) by $f(q)$, unless the number of data items requested from $\mathcal{D}_k$ will increase above $r(q)$. If that is the case (lines 11–12), we reduce actual move gain $g_m(d^*, k)$ by $f(q)$. If d^* is replicated (lines 13–17) we find the disk from which it is retrieved for query q (line 14). If the disk from which d^* is requested a bottleneck

disk for q and if the number of data items requested from that disk is larger than the optimal distribution of q, then we increase the actual replication gain of d^* to $\mathcal{D}_k$ by $f(q)$, unless $\mathcal{D}_k$ will become a bottleneck disk after this replication.

Algorithm 8: Update virtual leave gains after a move or replication from $\mathcal{D}_s$ to $\mathcal{D}_z$.

Require: $(\mathcal{D}, \mathcal{Q})$, $\Pi_K = \{\mathcal{D}_1, \ldots, \mathcal{D}_K\}$, $d^* \in \mathcal{D}_s$, *opType*

```
1: if opType = replication then
2:    replicate d* to D_z and lock
3: else
4:    move d* to D_z and lock
5: for each query q ∈ Q that contains d* do
6:    s ← getRequestedDisk(q, d*)
7:    if opType = replication ∧ t_s(q) ≤ t_z(q) then
8:       continue
9:    else
10:      setRequestedDisk(q, d*, z)
11:      t_s(q) ← t_s(q) − 1, t_z(q) ← t_z(q) + 1
12:      for each unlocked data item d ∈ q do
13:         m ← getRequestedDisk(q, d)
14:         if t_s(q) = r_opt(q) ∧ m = s then
15:            vg(d) ← vg(d) − f(q)
16:         if t_z(q) = r_opt(q) + 1 ∧ m = z then
17:            vg(d) ← vg(d) + f(q)
```

Algorithm 8 updates the virtual leave gains of unlocked data items that are neighbor to d^*, after moving or replicating d^* from $\mathcal{D}_s$ to $\mathcal{D}_z$. Parameter *opType* indicates whether we are performing a move or a replication. In lines 5–17, we check each query q that requests d^*. We first find the disk from which d^* is requested for query q (line 6). If d^* is being replicated and the number of documents requested from disk $\mathcal{D}_s$ for query q is less than that of $\mathcal{D}_z$, there is no need for update due to query q (lines 7–8). Otherwise, we set the optimal disk to retrieve d for q to $\mathcal{D}_z$ (line 10). If prior to the operation, the number of data items requested from $\mathcal{D}_s$ was larger than those of $\mathcal{D}_z$, we update the $t_s(q)$ and $t_z(q)$ values (line 11). The for loop in lines 12–17 updates the virtual leave gain of each data item d that is neighbor to d^*. If d is requested from $\mathcal{D}_s$ for q and the number of data items requested from $\mathcal{D}_s$ is equal to $r_{opt}(q)$, then virtual leave gain of d is reduced by $f(q)$ (lines 14–15). If d is requested from $\mathcal{D}_z$ for q and the number of data items requested from $\mathcal{D}_z$ is equal to $r_{opt}(q) + 1$, then the virtual leave gain of d is increased by $f(q)$ (lines 16–17).

2.7 Complexity Analysis

In this section, we provide detailed complexity analyses of the recursive replicated declustering and multi-way replicated refinement phases of our algorithm.

2.7.1 Recursive replicated declustering phase

In the recursive replicated declustering phase, initial gain computations (Algorithm 2) take $O(2 \times \sum_{q \in Q} |q|) = O(\sum_{q \in Q} |q|)$ time for each two-way refinement pass. This is because, each data item has two gains (either g_m and g_r or g_{u_A} and g_{u_B}), and for each data item d, we check all the queries by which d is requested to identify the initial operation gains of d. After selecting the best operation to perform and the related data item, we perform an extractMax for the selected operation, and delete the other operation gain related with the data item. Thus, at each pass, at most $|\mathcal{D}|$ extract-max and at most $|\mathcal{D}|$ heap-delete operations are performed. When an operation is performed on a data item, the operation gains of its neighboring data items must be investigated for possible updates (Algorithms 3, 4, 5). In the implementation of gain update operations, we use increase-key or decrease-key operations in max-heaps. If an operation is invoked on every data item during an FM-like pass, every query q incurs at most $|q|^2$ updates in total. Note that this $|q|^2$ upper bound is very loose since when an operation is performed on a data item d requested by a query q, only in a handful of conditions (that are determined by changes in Δ values of q) the update of the gain values of the other data items requested by q is necessary. Since we don't update the gains of previously iterated data items in a pass, after each operation, the maximum number of updates reduces by one.

When all operations are considered, in one two-way refinement pass, at most $(|q| \times (|q| - 1)/2) \times 2 \approx |q|^2$ gain update operations will have to be performed on the data items of a query q in the worst case. (Each data has two gains, hence the multiplication by two). Totally, it makes $O(\sum_{q \in Q} |q|^2)$ gain update operations in each two-way refinement pass. Thus, the gain update cost of a

two-way refinement pass is $O(\sum_{q \in Q} |q|^2 \times lg|\mathcal{D}|)$. The total cost of a two-way refinement pass is $O(|\mathcal{D}| \times lg|\mathcal{D}| + \sum_{q \in Q} |q|^2 \times lg|\mathcal{D}|) \approx O(\sum_{q \in Q} |q|^2 \times lg|\mathcal{D}|)$ for practical purposes. We limit the number of passes to 10 for a recursive replicated declustering step and in practice the number of passes rarely reaches to 10.

The worst-case analysis above is valid for the first two-way replicated declustering where there are no replications initially. We proceed with the analysis of the recursive replicated declustering by investigating the complexity in the levels of the recursion tree. We assume that maximum allowable replication occurs after the first two-way replicated declustering step, which is a worst case scenario. So at each recursion level the sum of the sizes of the sub-datasets will be at most $|D|(1 + r)$. Under a balanced declustering assumption, at the ℓth recursion level, two-way declustering processes will be applied on 2^ℓ sub-datasets each of size $O(|D|(1 + r)/2^\ell)$. If we ignore the decrease in the query sizes due to the query splitting process, the aggregate cost of 2^ℓ two-way declustering processes at the ℓth recursion level will be $O(2^\ell \times |D|(1 + r) \times \sum_{q \in Q} |q|^2)$, leading to an overall cost of $O(K \times lg(|\mathcal{D}|(1 + r)) \times \sum_{q \in Q} |q|^2)$. However, for practical purposes, considering the decrease in the query sizes due to the query splitting, the aggregate cost of 2^ℓ two-way declustering processes at the ℓth recursion level will approximately be $O(|D|(1 + r) \times \sum_{q \in Q} |q|^2)$, leading to an overall cost of $O(lg(K) \times lg(|\mathcal{D}|(1 + r)) \times \sum_{q \in Q} |q|^2)$.

2.7.2 Multi-way replicated refinement phase

Multi-way replicated refinement phase has a preprocessing step where we compute the optimal schedules for all queries. This preprocessing takes $O(\sum_{q \in Q}(|q|^2 \times K)$ time.

The complexity analysis of each multi-way refinement pass is as follows: In initial virtual leave gain computation, since we have a single gain for each data item, the complexity is similar to the initial gain computation of a two-way replicated declustering. The only difference roots from the $getRequestedDisk(q, d)$ operation that takes $O(|q|)$ time. Thus, calculating the virtual gains of every item takes

$O(\sum_{q \in Q} |q|^2)$ time. We build a heap from the virtual leave gains in $O(|\mathcal{D}|)$ time. We select the data item with the maximum virtual leave gain in $O(log(|\mathcal{D}|)$ time. Then we compute its actual move and replication gains for the remaining $K-1$ disks and decide where to move or replicate according to this actual gains. In total, actual gain calculations for all data items take $O(\sum_{q \in Q} |q| \times K)$ time. After performing the operation that has the highest actual gain, we update the virtual leave gains of neighboring data items as well as the item distributions of all related queries. Updating virtual leave gain of a data item takes $O(lg|\mathcal{D}|)$ time, since we store virtual leave gains in a max-heap. Similar to the recursive replicated declustering phase, we may need to update the virtual leave gains at most $|q|^2$ times in a multi-way refinement pass, leading to a $O(\sum_{q \in Q} |q|^2 \times lg|\mathcal{D}|)$ complexity. In the worst case, the total cost of virtual leave gain updates is $O(\sum_{q \in Q} |q|^2 \times lg|\mathcal{D}|)$. Updating query item distributions takes a total of $O(\sum_{q \in Q} |q|)$ time. Thus, in a multi-way refinement pass, the total cost for updates is $O(\sum_{q \in Q} |q|^2 \times lg|\mathcal{D}|)$.

We limited the number of passes in multi-way refinement to 10, thus, when we take the preprocessing, virtual leave gain initialization, actual gain computation and virtual leave gain update stages into account, the total cost for multi-way refinement is $O(\sum_{q \in Q} |q|^2 \times K) + O(\sum_{q \in Q} |q|^2) + O(\sum_{q \in Q} |q| \times K) + O(\sum_{q \in Q} |q|^2 \times lg|\mathcal{D}|)$, which is equal to $O((K + log|\mathcal{D}|) \times \sum_{q \in Q} |q|^2)$.

2.8 Experimental Results

In this section, we present the results of experiments conducted to compare the performance of the proposed Selective Replicated Assignment (SRA) scheme against the state-of-the-art Random Duplicate Assignment (RDA) and Orthogonal Assignment (OA) schemes. RDA and OA are selected since they are known to perform good for arbitrary queries. Also it is possible to modify these approaches for selective replication. We modified both RDA and OA to support partial replication, and improved RDA such that it utilizes query logs and selects the most frequently requested data items and replicates them at random disks. We call this modified version the Most Frequent Assignment (MFA) scheme.

In our comparisons we used 9 datasets: *Airport*, *Bea*, *Face*, *FR*, *HH*, *Ntar*, *Park*, *Place90*, and *State*. The properties of these datasets are presented in Table 2.2. The datasets are taken from [3] and divided into 4 groups. As described in [3], the *Face* dataset is a collection of gray-scale face images containing 144 images from the MIT image database [26], 300 images from PEIPA [27] and 400 images from the ORL image database [28]. These 844 images are used to construct an image retrieval system using the algorithm described in [24]. In this algorithm, the significant pixels of the images are extracted by multi-resolution wavelet analysis and a number of significant pixels are kept as signature for each image. Thus, each pixel location defines a relation among the images (data items) that contain the pixel in their signature files. As a query is a signature (i.e., a set of pixel locations), the set of all possible pixel locations in the images constitutes the query set.

The second group of datasets consists of multi-feature point data used for function-approximation experiments [29]. The *HH* and *FR* datasets contain 22784 and 40768 points in 16 and 10 dimensions, respectively. These datasets are indexed into a grid directory with cell size restricted to 16 points as described in [30]. The resulting grid directory contains 1638 data pages (data items) for *HH* and 3338 data pages for *FR*. A set of synthetic rectangular and diagonal queries is generated assuming Gaussian distribution for both query sides and centers for each dataset.

Other datasets consist of GIS data collected from the National Transportation Atlas Databases [31]. The *Airport* and *Place90* datasets contain two-dimensional point data. *Airport* contains the public use airports and landing facilities in the US. *Place90* contains place locations from the 1990 Census Master Area reference file. Airport, containing 6735 points, is indexed into a grid file of 1176 pages with cell capacity of 8 points. Similarly, *Place90*, containing 23651 points, is indexed into a grid file of 3382 pages. The *Park*, *Ntar*, *State* and *Bea* datasets contain two-dimensional polygon data. The bounding box of every polygon is considered as a data item for these datasets. *Park* contains the national parks, *Ntar* contains the national transportation analysis regions, *Bea* contains the economic areas, and State contains the US boundaries with integrated shorelines. A set of synthetic

Table 2.2: Properties of datasets.

Class	$\mathcal{D}$ Dataset	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{Q}\rvert$	Average query size
Image	*Face*	844	1024	23.1
Func. Approx.	*HH*	1638	2000	43.3
	FR	3338	10000	10.0
GIS (Point)	*Airport*	1176	5000	22.8
	Place90	3382	12000	17.9
GIS (Polygon)	*Park*	1022	4000	20.1
	Ntar	8952	10000	29.2
	Bea	10674	20000	26.8
	State	10827	10000	33.5

rectangular and diagonal queries are generated for the GIS datasets as for the function-approximation datasets.

While testing the performance of MFA and SRA, the query sets for all datasets except *Face* are divided into two equal parts. The first half is used for replication and declustering and the second half is used for testing the performance. The query set for *Face* is composed of all possible queries so it is fully used while declustering and testing of *Face*.

All of the algorithms used in the experiments are implemented in C programming language, and experiments are conducted on a 2GHz Intel Core Duo machine with 2MB L2 cache and 2GB DDR2 667 MHz memory.

Query processing performances of the compared algorithms are tested on K=16, 24, 32 disks and the allowed overall replication ratio is varied from 10% to 100%. With 9 different datasets, 3 different disk counts, and 10 different replication ratio values, we present the results of 270 different experiment instances. For each SRA experiment instance, we report the average of 10 runs, since we use randomly generated initial feasible two-way declusterings in our replicated declustering phase.

The query processing performance of a given algorithm is evaluated in terms

of the average retrieval overhead per query induced by the resulting replicated declustering. Here, average retrieval overhead per query (arO) for a given replicated declustering of a dataset and a query set is defined as total response time overhead (Equation 2.5) divided by the number of queries. That is,

$$arO(Q) = TrO(R_K, Q)/|Q|. \tag{2.17}$$

Table 2.3: Arithmetic averages of the arO values for K=32 disks over the nine datasets.

	percent distribution of replication ratio among recursive replicated declustering and multi-way refinement phases						
	w.out unrep.	with unreplication					
% rep.	100–0	100–0	80–20	60–40	40–60	20–80	0–100
10%	0.40	0.31	0.29	0.27	0.24	0.23	0.21
20%	0.36	0.28	0.25	0.22	0.19	0.16	0.15
30%	0.32	0.22	0.23	0.19	0.15	0.12	0.11
40%	0.25	0.19	0.20	0.18	0.12	0.09	0.09
50%	0.19	0.15	0.15	0.14	0.10	0.06	0.07
60%	0.15	0.12	0.13	0.13	0.09	0.05	0.06
70%	0.12	0.09	0.11	0.11	0.09	0.04	0.05
80%	0.10	0.07	0.09	0.09	0.08	0.03	0.04
90%	0.07	0.05	0.07	0.07	0.08	0.03	0.03
100%	0.06	0.04	0.06	0.06	0.07	0.02	0.02

In Table 2.3, we present the arithmetic averages of the average retrieval overhead of SRA over the nine datasets with increasing replication ratio, where the allowed replication ratio is distributed between the recursive replicated declustering and multi-way refinement phases according to the percentage values displayed over the columns. For example, the column header 80–20 indicates that the recursive replicated declustering phase is allowed to utilize 80% of the replications and the multi-way refinement phase is allowed to utilize 20% of the replications. The values in the table indicate the retrieval overhead of the replicated declusterings obtained by SRA under the given replication distribution.

The second column of Table 2.3 is introduced to justify the usage of unreplication operation in recursive replicated declustering phase. Note that the

100%–0% replication-distribution scheme provides an approach where replication is only performed in recursive replicated declustering phase. The third and second columns of Table 2.3 show the performance of such a system where unreplication operation is utilized and not utilized, respectively. By comparing these two columns we can observe that embedding unreplication operation always improves the performance of the recursive replicated declustering phase.

As seen in Table 2.3, especially for low replication ratios (between 10% to 30%), the average results obtained by SRA are best when the given replication amount is fully utilized in the multi-way refinement phase (0%–100% replication-distribution). However, for higher replication ratios (between 40% to 100%), best results are obtained in the 20%–80% replication-distribution scheme. These results indicate that, for small allowed replication ratios, performing replications at a later phase, that is during the K-way declustering phase, brings more gain, whereas for higher allowed replication ratios, performing a small percent of the replications at an earlier phase, that is during the recursive bipartitioning phase, has a more positive effect on the overall SRA performance. Since 20%–80% replication-distribution scheme has better results for more experiment instances, the results reported for SRA in the following figures are obtained with this replication-distribution scheme. The good results observed for the 0%–100% and 20%–80% replication-distribution schemes point to the success of our multi-way replicated refinement algorithm. The fact that 20%–80% replication-distribution scheme, which is a combination of recursive replicated declustering and multi-way replicated refinement schemes most of the time outperforms the 0%–100% scheme, which is an approach where replication and declustering is decoupled demonstrates the need for our recursive replicated declustering algorithm.

Figs. 2.5 and 2.6 display the individual performances of the algorithms over each of the nine datasets for $K = 16, 24, 32$. In the figures, variation of the arO values of algorithms with increasing replication ratio is presented. Closer points to x-axis mean better average retrieval times.

As seen in Figs. 2.5 and 2.6, SRA has better (smaller) average retrieval time than MFA and OA for all experiment instances. While comparing MFA with

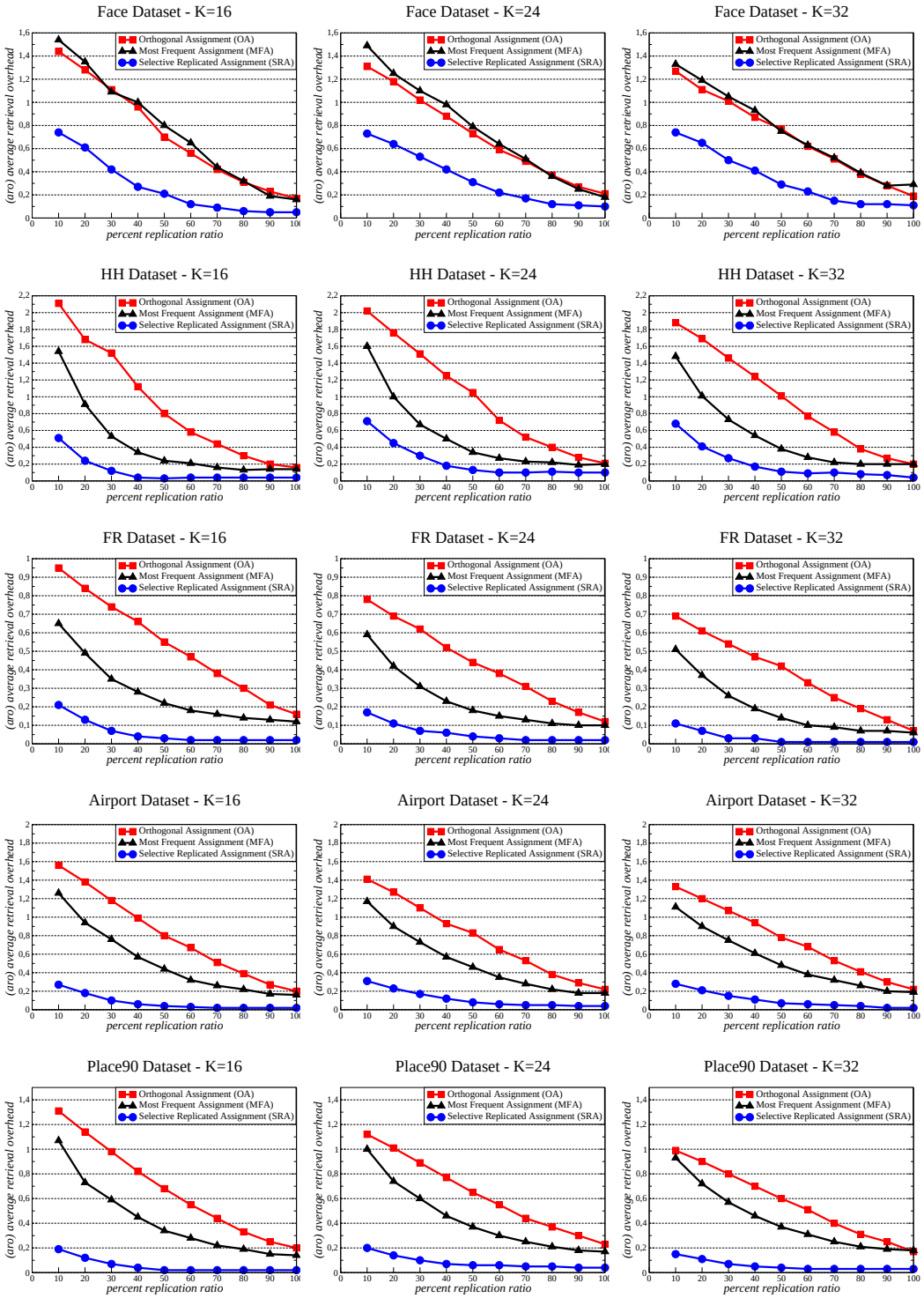


Figure 2.5: Average retrieval overhead vs replication ratio figures for the *Face, HH, FR, Airport, Place90* datasets

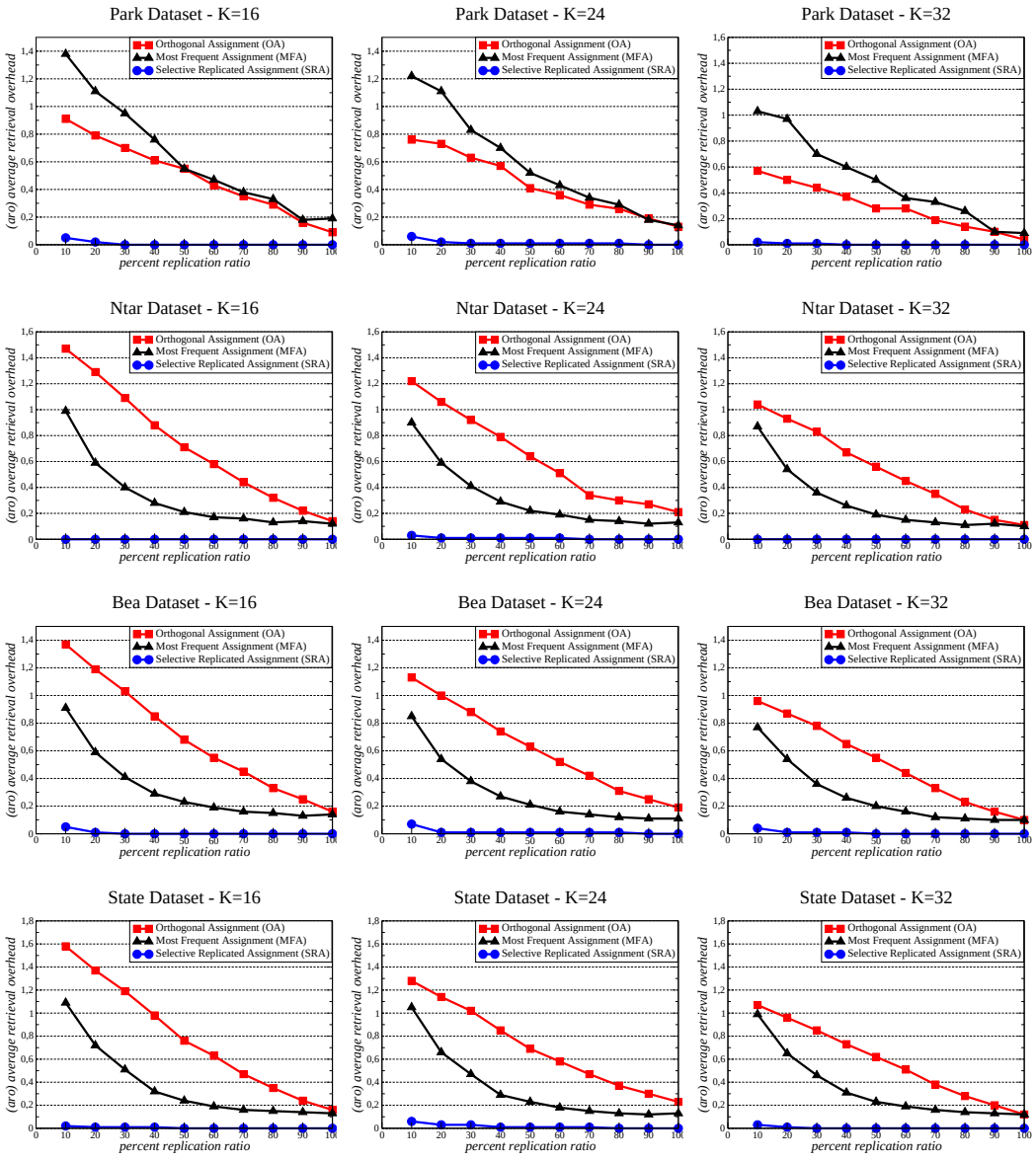


Figure 2.6: Average retrieval overhead vs replication ratio figures for the *Park*, *Ntar*, *Bea*, *State* datasets

OA, MFA performs much better than OA in seven of the nine datasets. Only in *Face* and *Park* datasets OA performs slightly better than MFA. We observe that with increasing replication amount, the deviation of OA from the strictly optimal declustering decreases linearly, whereas in both MFA and SRA we observe a quadratic decrease. These results point to the importance of using query logs in improving performance, since MFA also makes use of query logs by replicating frequently requested items.

As seen in Figs. 2.5 and 2.6, the performance gap between the proposed SRA algorithm and the existing MFA and OA algorithms decreases with increasing replication amount. But SRA performs considerably better than the other algorithms even for 100% replication.

As seen in Figs. 2.5 and 2.6, SRA has better (smaller) average retrieval time than MFA and OA for all experiment instances. While comparing MFA with OA, MFA performs much better than OA in seven of the nine datasets. Only in *Face* and *Park* datasets OA performs slightly better than MFA. We observe that with increasing replication amount, the deviation of OA from the strictly optimal declustering decreases linearly, whereas in both MFA and SRA we observe a quadratic decrease. These results point to the importance of using query logs in improving performance, since MFA also makes use of query logs by replicating frequently requested items. An analysis of the Figs. 2.5 and 2.6 reveals that the performance gap between the proposed SRA algorithm and the state-of-the-art MFA and OA algorithms decreases with increasing replication amount. However, as also seen in the figure, SRA still performs considerably better then MFA and OA even for high replication amounts.

An analysis of the arithmetic averages of the average retrieval overheads and the running times of the MFA, OA and SRA over the nine datasets with increasing replication ratio reveals that, for K =16, 24 and 32 disks, even with low replication ratios such as 10%, SRA achieves very low overheads and to achieve similar overheads MFA requires around 70%, whereas OA requires around 90% replication.

Table 2.4 shows the standard deviation values computed over the costs of

Table 2.4: Standard deviation values of SRA over the 9 datasets.

	% replication	*Face*	HH	FR	*Airport*	Place90	Park	*Ntar*	Bea	State
	10%	0.54	0.45	0.15	0.35	0.24	0.23	0.07	0.31	0.14
	20%	0.49	0.29	0.10	0.24	0.18	0.05	0.00	0.06	0.01
	30%	0.36	0.15	0.06	0.17	0.12	0.01	0.02	0.02	0.02
	40%	0.26	0.06	0.03	0.10	0.06	0.02	0.01	0.02	0.02
K=16	50%	0.18	0.03	0.03	0.07	0.05	0.00	0.01	0.02	0.02
	60%	0.12	0.07	0.02	0.04	0.03	0.00	0.01	0.02	0.00
	70%	0.08	0.04	0.02	0.03	0.03	0.00	0.01	0.02	0.01
	80%	0.08	0.08	0.03	0.03	0.04	0.00	0.00	0.01	0.01
	90%	0.08	0.06	0.03	0.03	0.04	0.03	0.00	0.01	0.00
	100%	0.04	0.06	0.02	0.03	0.03	0.03	0.00	0.01	0.00
	10%	0.43	0.52	0.09	0.29	0.18	0.20	0.25	0.39	0.28
	20%	0.40	0.42	0.07	0.24	0.14	0.06	0.09	0.12	0.13
	30%	0.40	0.30	0.05	0.18	0.12	0.00	0.12	0.09	0.18
	40%	0.30	0.25	0.04	0.14	0.08	0.06	0.03	0.14	0.12
K=24	50%	0.26	0.17	0.03	0.10	0.06	0.02	0.02	0.12	0.03
	60%	0.15	0.15	0.02	0.08	0.05	0.02	0.03	0.07	0.04
	70%	0.12	0.14	0.00	0.05	0.05	0.02	0.02	0.05	0.04
	80%	0.09	0.16	0.02	0.06	0.05	0.03	0.01	0.03	0.04
	90%	0.09	0.12	0.01	0.05	0.04	0.05	0.02	0.03	0.02
	100%	0.06	0.14	0.02	0.06	0.06	0.04	0.01	0.03	0.01
	10%	0.34	0.44	0.03	0.21	0.11	0.12	0.06	0.25	0.18
	20%	0.34	0.36	0.02	0.16	0.09	0.02	0.02	0.08	0.08
	30%	0.31	0.22	0.02	0.13	0.07	0.00	0.09	0.10	0.14
	40%	0.21	0.14	0.00	0.08	0.06	0.03	0.03	0.15	0.09
K=32	50%	0.22	0.12	0.00	0.06	0.04	0.00	0.03	0.14	0.03
	60%	0.15	0.12	0.00	0.04	0.04	0.00	0.01	0.11	0.03
	70%	0.10	0.10	0.00	0.06	0.04	0.00	0.01	0.04	0.03
	80%	0.06	0.14	0.00	0.03	0.02	0.00	0.00	0.04	0.02
	90%	0.10	0.11	0.01	0.04	0.04	0.00	0.01	0.03	0.02
	100%	0.07	0.12	0.00	0.06	0.02	0.02	0.01	0.03	0.02

individual queries for the datasets when SRA is used. As seen in the table, the confidence interval of the results produced by SRA becomes narrower as the replication ratio increases, which is expected since with high replication ratios the declusterings found by SRA approaches to optimal ones. By investigating Fig. 2.5, Fig. 2.6, and Table 2.4 we can observe that even the confidence interval of the solutions produced by SRA remain below the averages of the solutions produced by MFA and OA.

Chapter 3

Re-Declustering for Elasticity in Parallel Databases

Due to fast changing dynamics of many applications, ever increasing data storage requirements, and big data processing economics, parallel database systems are forced to migrate into cloud computing settings, an environment which demands support for elasticity from its tenants. Supporting changes in server counts, which may invalidate the previously computed declustering solutions, can also enforce the re-computation of the declustering solution. The re-declustering problem can be observed even in conventional parallel database settings. Due to high failure rates of conventional disks and commodity servers, constantly checking the integrity levels of disks and removing/replacing malfunctioning disks might be necessary or because of the fast increase rates of data stored in parallel database systems, database admins may need to introduce new servers in order to increase both capacity and performance of the system.

In Chapter 2, we proposed a replicated declustering scheme that utilizes query logs to selectively replicate and distribute data items and and show that this scheme outperforms existing replicated declustering schemes in the presence of query logs and especially under low replication constraints. Similarly, there are a

number of declustering and replicated declustering solutions relying on optimizations utilizing query logs [4, 32, 3, 33, 34]. However, in an approach that is based on query logs, due to possible changes of query patterns with time, the initial declustering solution may become obsolete. A re-computation of the declustering solution (*re-declustering*) might be necessary. Furthermore, realizing this newly computed re-declustering requires data migrations.

To recount, all the above mentioned three reasons, namely:

a. changes in data access patterns (significant pattern changes in query logs),

b. planned server removal due to reducing demand or for maintenance reasons, and

c. planned server addition to increase capacity and/or performance of the system

may enforce a re-declustering of the data items.

The re-declustering problem can be solved via running the replicated declustering scheme from scratch. However, in parallel disk systems often the sizes of the stored data items are quite large and thus, such a "scratch-remap" method that does not respect the existing disk placement of data items can cause unacceptable amounts of data migrations. Note that, performing a declustering process from scratch is not costly in itself since this computation is performed in memory, however, realizing the new mapping dictated by this new declustering can be quite time consuming and applications may suffer significantly during this long remapping process, as data migrations are quite costly.

In this chapter, we present the following contributions:

- We propose a novel weighted bipartite matching model that given a dataset, a newly proposed replicated declustering solution for this dataset, and an existing replicated mapping of the data items in the dataset to the servers of the parallel database system, provides a matching of the overlapping subsets

in the replicated declustering solution to servers such that the number of data item migrations necessary to realize the new solution is minimized. Proposed model can compute the migration minimizing matching optimally even when the number of overlapping subsets and the number of servers are different, thus supporting server additions or removals.

- We also propose a novel abstraction scheme that enables the encoding of migration operations necessary to realize the new solution as queries. Through this abstraction, given a newly proposed replicated declustering solution, a query workload to be performed over the system, and and an existing (old) replicated declustering solution, it becomes possible to reduce query processing and migrations costs together. This abstraction also enables the use of a state-of-the-art replicated declustering scheme to be used for the solution of re-declustering problem with minor extensions.

- We propose extensions to an efficient and effective iterative improvement scheme that improves query processing costs of a replicated declustering solution so that this extended scheme not only improves query processing costs but also reduces migration amounts as well.

- We propose a three-phase log-utilizing replicated re-declustering scheme that suggests a new replicated declustering solution over any existing replicated declustering solution such that the new solution strikes a balance between the conflicting objectives of reducing query processing costs and reducing migration costs. The proposed re-declustering scheme enables optimizations to query processing costs when changes in query-log patterns, disk additions, or disk removals are performed, while considering migration costs.

The organization of this chapter is as follows. In Section 3.1, we present the notations used in this chapter and the problem definition. Proposed re-declustering algorithm is presented in Section 3.2. Section 3.3 presents and discusses the conducted experiments where we compare the SRA algorithm proposed in Chapter 2 with the re-SRD algorithm proposed in this chapter. We provide a brief overview of the related literature in Section 3.4.

Table 3.1: Notations used in this chapter in addition to those presented in Table 2.1.

Symbol	Description
S_k	Server k
$\mathcal{S}_k$	Data items in server S_k
$RSP(\mathcal{D})$	A Replica-to-Server Placement strategy for the data items in $\mathcal{D}$
$rss(d)$	Replica-Server-Set for data item d, i.e., the set of servers holding a replica of d

3.1 Notations and Problem Definition

Just like in Chapter 2, we assume a dataset $\mathcal{D}$ and a query set Q with $|\mathcal{D}|$ indivisible data items and $|Q|$ queries, respectively, and a query $q \in Q$ is a subset of $\mathcal{D}$. Again, each query q has a frequency $f(q)$ that shows the number of times q is requested in Q. We also assume that all server disks are homogeneous and the retrieval time of all data items on all disks are equal and can be accepted as one for practical purposes. The notations used in this chapter are presented in Table 3.1. As you can see in the table, we introduced the $RSP(\mathcal{D})$: A Replica-to-Server Placement strategy for the data items in $\mathcal{D}$ and $rss(d)$: Replica-Server-Set for data item d, in other words, the set of servers holding a replica of d notations in addition to the other notations presented in Chapter 2. We also introduced the S_k: Server k, notation to distinguish parallel database servers from each other. These notations are introduced since unlike in Chapter 2, in this chapter, the placement of data subsets to different servers cause different migration costs.

Definition *Replicated Re-Declustering Problem*: Given a set $\mathcal{D}$ of data items, a set Q of queries, K homogeneous servers each with a storage capacity of C_{max}, a replica-to-server placement scheme $RSP_{old}(\mathcal{D})$ showing the current mapping of data items in $\mathcal{D}$ to the servers, and a maximum allowable replication ratio r, the *Replicated Re-Declustering Problem* tries to find a new replica-to-server placement scheme $RSP_{new}(\mathcal{D})$ subject to the server capacity constraints with the objective of minimizing a metric defined over the total parallel response time of Q

and the total amount of data item migrations needed to realize the new mapping $RSP_{new}(\mathcal{D})$.

Depending on the reason for re-declustering we may have the following three scenarios for the server counts of the new replica-to-server placement scheme:

a. A re-declustering due to query log changes will be performed. In this scenario, the number of servers in the old and new placement schemes are equal.

b. A re-declustering due to the removal of a set of servers will be performed. Assuming that the set of servers to be removed is $\mathcal{S}^{rem}$, and the number of servers to be removed is $|\mathcal{S}^{rem}| = \Delta K$, the number of servers in the new placement schemes is equal to $K - \Delta K$.

c. A re-declustering due to addition of a set of servers will be performed. Assuming that the set of servers to be added is $\mathcal{S}^{add}$, and the number of servers to be added is $|\mathcal{S}^{add}| = \Delta K$, the number of servers in the new placement scheme is equal to $K + \Delta K$.

Note that in the *Replicated Re-Declustering Problem* there are two conflicting objectives, namely the minimization of total parallel response time and the minimization of total data item migrations. As with most multi-objective problems, a solution to this problem must also strike a balance between these two conflicting objectives. We also provide a mechanism to fine-tune the balance between these two objectives and shall further discuss this mechanism in Section 2.4.2.

3.2 Proposed Approach

In order to address the K-way replicated re-declustering problem, we propose a three-phase approach. In the first phase, a recursive replicated declustering heuristic, which reduces query processing times for queries without paying any attention to migration costs, is utilized to divide the data items in the dataset

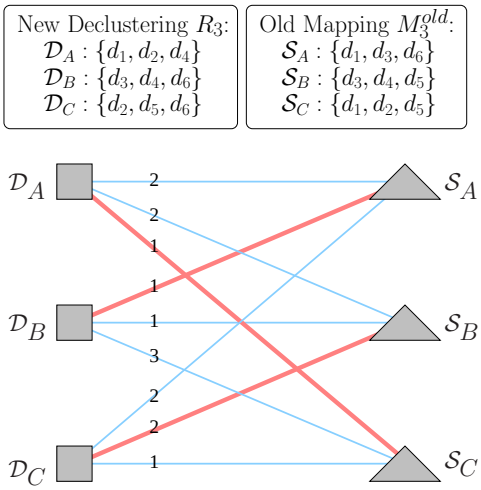
$\mathcal{D}$ into desired number of ($K - \Delta K$, K, or $K + \Delta K$) possibly overlapping subsets. In the second phase, a one-to-one data-subset-to-server mapping scheme that maps the subsets obtained at the end of the first phase to the servers is proposed. This mapping is performed in a way that minimizes the total amount of data migrations. Here, we show how to formulate the one-to-one data-subset-to-server mapping problem as a weighted bipartite matching problem, which we solve optimally. In the third phase, a multi-way migration-aware replicated refinement heuristic is used to further reduce both the query processing times and the amount of necessary migrations.

We should remind here that until the end of third phase no actual data item migration is performed. Migrations are realized only after all the computations are performed in memory and the final replica-to-server placement $RSP_{new}(\mathcal{D})$ is computed.

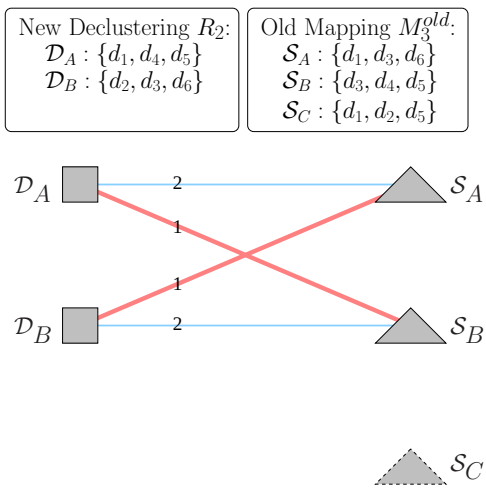
3.2.1 Initial Replicated Declustering Phase

In this phase, our aim is find a "good" replicated declustering solution for the given query workload and the desired number of servers. The previous mapping and the previous number of disks are disregarded in this phase. Assuming that K is the target number of servers in the new setting, at the end of this phase, K possibly overlapping subsets of the dataset $\mathcal{D}$, say $R_K = \{\mathcal{D}_1, \ldots, \mathcal{D}_K\}$ is obtained. To this end, we utilize the recursive replicated declustering schema proposed in [34], since this scheme is known to successfully reduce query response costs of queries in the presence of query logs. However, it is possible to use any replicated declustering scheme in this phase.

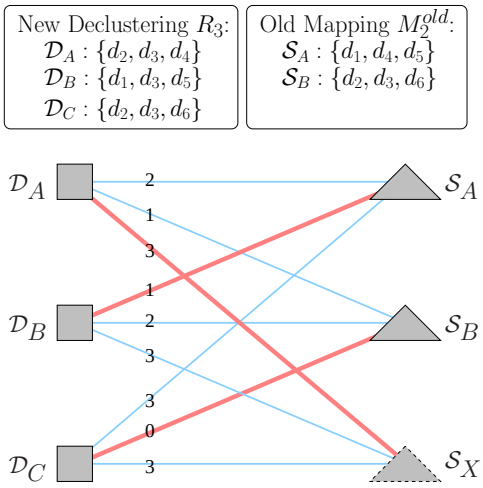
In the recursive replicated declustering schema proposed in [34], a two-way replicated declustering algorithm is recursively applied to obtain the desired number of overlapping subsets. The two-way replicated declustering algorithm starts from a given initial randomly generated two-way declustering, and iteratively improves this two-way declustering via *move*, *replication* and *unreplication* operations so as to optimize the query processing times of the queries in Q.



(a) A new replicated declustering due to query changes is performed. Weighted bipartite graph model for mapping the parts of the new declustering to disks. The weight of the matching is 1+1+2=4.



(b) Removal of server $\mathcal{S}_C$ from a three server system will be performed. Since d_2 resides only in $\mathcal{S}_C$, it has to be migrated to the server that matches $\mathcal{D}_B$. The weight of the matching is 1+1=2.



(c) Addition of a new server ($\mathcal{S}_X$) to a two server system will be performed. Note that since $\mathcal{S}_X$ is newly introduced, all of its edges has a cost of three. The weight of the matching is 3+1+0=4.

Figure 3.1: Weighted bipartite matching model indicates how many migrations are necessary to realize the new declustering. For example in (b), to realize the new declustering, d_2 has to migrate from S_C to S_A and d_1 has to migrate from S_A to S_B. The numbers on the edges indicate edge weights and the matchings are indicated with thick edges.

3.2.2 Minimum Weighted Bipartite Matching

In this phase we propose a one-to-one data-subset-to-server mapping scheme that maps the K overlapping subsets ($R_K = \{\mathcal{D}_1, \ldots, \mathcal{D}_K\}$) generated in the first phase to the servers. This mapping is done considering the old replica-to-server placement scheme $RSP_{old}(\mathcal{D})$, which shows the current distribution of data items among the servers. For the clarity of discussions let us assume that S_k indicates the set of data items stored in server S_k. That is, $S_k = \{d_i : S_k \in rss(d_i)\}$.

A mapping of an overlapping subset $\mathcal{D}_x$ to a server S_y implies that S_y will store and serve the data items in the subset $\mathcal{D}_x$. This may require some data items to be replicated to S_y and the objective is to find the mapping that requires the minimum amount of replications. We perform this mapping by first formulating the this problem to the minimum weighted bipartite matching (MWBM) problem, and then by retrofitting the solution of the MWBM as a mapping. The optimal solution to MWBM also minimizes the amount of data migrations incurred by the current mapping. The optimal solution to MWBM is computed using the network flow codes developed from [35].

The complete bipartite graph $G = (\mathcal{V}_D, \mathcal{V}_S, E)$ that is used for matching is constructed as follows. We are given the overlapping subsets obtained in the first phase and the existing mapping of data items to servers. For each overlapping subset $\mathcal{D}_i \in R_K$ there is a vertex in $\mathcal{V}_D$ and for each server S_j in the system there is a vertex in $\mathcal{V}_S$. Each vertex of the first set is connected to every vertex of the second set, thus the bipartite graph is complete. The edge cost of the edge between the vertex of an overlapping subset $\mathcal{D}_i$ and the vertex of a server S_j is set to the number of data item migrations necessary to make the contents of server S_j same with to the contents of overlapping subset $\mathcal{D}_i$. That is,

$$cost(edge\{\mathcal{D}_i, S_j\}) = |\mathcal{D}_i \backslash \mathcal{S}_j|. \tag{3.1}$$

After generating the complete bipartite graph G, we solve the MWBM problem. The obtained minimum weighted perfect matching $\mathcal{M} = \{(v_i, v_j) : v_i \in \mathcal{V}_D \wedge v_j \in \mathcal{V}_S\}$ between the vertex sets $\mathcal{V}_D$ and $\mathcal{V}_S$ induces an overlapping-subset-to-server mapping and the cost of this matching indicates the amount of migration to be

performed due to this mapping. In other words, if $(v_i, v_j) \in \mathcal{M}$, then this implies that vertex set $\mathcal{D}_i$ is mapped to server S_j.

The construction of the MWBM model for the three different re-declustering scenarios we mentioned before are further explained with examples in the following sections. In the example bipartite graphs, for each overlapping subset $\mathcal{D}_i$ there is a vertex at the left side (indicated with squares) and for each server S_j there is a vertex at the right side (indicated with triangles). For the clarity of presentation, the examples in the following sections show addition or removal of single servers. However, extending our matching schemes to support addition or removal of multiple servers is quite straightforward and how these extensions might be performed are briefly discussed at the end of each subsection.

3.2.2.1 Data item access patterns change

In this scenario, it is assumed that a re-declustering due to query log changes is being performed and thus the number of servers in the old mapping and the number of overlapping subsets that is obtained at the end of first phase are equal. Fig. 3.1(a) demonstrates an example for the MWBM model used for the case where a re-declustering is performed due to query log changes. We are given the overlapping subsets $R_3 = \{\mathcal{D}_A, \mathcal{D}_B, \mathcal{D}_C\}$ obtained in phase one and the data item distribution of the old declustering M_3^{old}. The edge costs are determined according to Eq. 3.1. For example, the cost of edge $(\mathcal{D}_A, S_B)$ is 2 since if we were to map $\mathcal{D}_A$ to S_B, to realize this mapping, we would need to migrate d_1 and d_2 to S_B. The computed minimum weighted matching is marked with bold edges in the example and for this case has a total cost of four.

3.2.2.2 Removal of server S_k

In this scenario, it is assumed that a re-declustering due to a server removal is being performed and thus the number of servers in the old declustering (K) is one more than the number of overlapping subsets ($K - 1$) that is obtained at

the end of first phase. In the complete bipartite graph $G = (\mathcal{V}_D, \mathcal{V}_S, E)$, for each overlapping subset $\mathcal{D}_i \in R_{K-1}$ there is a vertex in $\mathcal{V}_D$ and for each server $S_j \neq S_k$ there is a vertex in $\mathcal{V}_S$.

Fig. 3.1(b) demonstrates an example for the MWBM model used for the planned removal of a server. In this example, we assume that server S_C will be removed from the system. We are given the overlapping subsets $R_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$ obtained by the new declustering (computed in phase one) and the data item distribution of the old declustering M_3^{old}. In the bipartite graph, for each overlapping subset $\mathcal{D}_i \in R_2$ there is a vertex at the left side and for each server S_j in the system there is a vertex at the right side. The edge costs are determined according to Eq. 3.1. For example, the cost of edge $(\mathcal{D}_B, S_B)$ is 2 since if we were to map $\mathcal{D}_B$ to S_B, to realize this mapping, we would need to migrate d_2 and d_6 to S_B. The computed minimum weighted matching has a total cost of two.

Note that for removal of ΔK servers, the number of overlapping subsets obtained at the end of first phase would be $K - \Delta K$ and in the bipartite graph model just not adding vertices for the removed servers suffices. At first look, not adding vertices for the removed servers is counter intuitive since there may be data items residing only in these servers. However, since the overlapping subsets obtained at the end of first phase covers all the data items and due to the edge cost definition, the migration costs of these data items are accounted for.

3.2.2.3 Addition of a server

In this scenario, it is assumed that a re-declustering due to a server addition is being performed and thus the number of servers in the old declustering, let us say K, is one less than the number of overlapping subsets that we is obtained at the end of first phase. In the complete bipartite graph $\mathcal{G} = (\mathcal{V}_D, \mathcal{V}_S, \mathcal{E})$, for each overlapping subset $\mathcal{D}_i \in R_{K+1}$ there is a vertex in $\mathcal{V}_D$ and for each server S_j in the system there is a vertex in $\mathcal{V}_S$. In addition, there is one more pseudo-vertex in $\mathcal{V}_S$ that represents the server to be added to the system. The edge costs are determined according to Eq. 3.1. All edges of the pseudo-vertex for S_X are

weighted with the number of elements in the respective overlapping subsets, since those many items will have to be migrated to S_X in case of a matching.

Fig. 3.1(c) demonstrates an example for the MWBM model used for server addition. In this example, we assume that server S_X will be added to the system. We are given the overlapping subsets $R_3 = \{\mathcal{D}_A, \mathcal{D}_B, \mathcal{D}_C\}$ obtained in phase one and the data item distribution of the old declustering M_2^{old}. The edge costs are determined according to Eq. 3.1. For example, the cost of edge $(\mathcal{D}_C, S_B)$ is set to 0 since if we were to map $\mathcal{D}_C$ to S_B, to realize this mapping, we'd not need to migrate any data items. The computed minimum weighted matching has a cost of four. Also note that all edges of the pseudo vertex for the newly added server S_X has a weight of 3 since S_X is assumed to be empty and all the overlapping subsets has a size of 3.

Note that for addition of ΔK servers, the number of overlapping subsets obtained at the end of first phase would be $K + \Delta K$ and in the bipartite graph model just adding ΔK pseudo vertices with appropriate weighting to the vertex set $\mathcal{V}_S$ of the bipartite graph suffices.

3.2.3 Migration-aware multi-way replicated refinement

In this section we propose a multi-way replicated refinement scheme that improves both query processing and migration costs. Note that there is a trade-off between the objective of minimizing data migration costs and the objective of minimizing query processing costs. However, basically they both try to minimize the costs associated with I/O operations, and hence, assuming unit data item sizes, the costs associated with migration and query processing can be combined and minimized within the same optimization objective.

In order to perform the optimization of both objectives in coordination, we propose an abstraction scheme that enables us to represent migration operations as queries. The proposed scheme is as follows. For each server S_k, we introduce a pseudo data item $d_{\mathcal{S}_k}$ (in total we introduce K pseudo data items). These pseudo

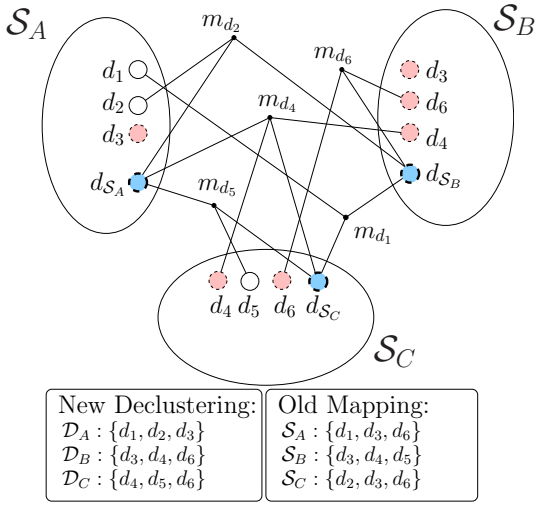


Figure 3.2: A sample 3-way replicated declustering and the newly added server data items ($d_{\mathcal{S}_A}, d_{\mathcal{S}_B}, d_{\mathcal{S}_C}$) and migration queries ($m_{d_1}, \ldots, m_{d_6}$).

data items are placed into their respective servers and they cannot be moved or replicated (they are fixed). Then for each data item d_i, we introduce a pseudo query m_{d_i}, which is used to model the migration costs associated with d_i. The pseudo query m_{d_i} is assumed to be requesting all replicas of d_i and all pseudo data items corresponding to the servers that *do not* have a replica of d_i in the old mapping. No matter the number of data items requested by m_{d_i}, the optimal response time $r_{opt}(m_{d_i})$ of m_{d_i} is set to one. The pseudo data items and migration queries are defined such that, if a migration query m_{d_i} requests more than one data item from a server S_k, than this means that d_i must be migrated (replicated or copied) to server S_k.

Fig. 3.2 shows a sample 3-way replicated declustering and the newly added migration queries and server data items. For the clarity of the figure, we do not present the regular queries in this figure. In the figure, there is a migration query for each data item and the data items that these queries demands are set according to the procedure explained above. For example, the migration query m_{d_5} is connected to d_5 and the pseudo vertices $d_{\mathcal{S}_A}$ and $d_{\mathcal{S}_C}$ for servers S_A and S_C, since these servers do not contain data item d_5 in the old mapping. Note that in Fig. 3.2 we did not illustrate migration query m_{d_3} as well, since it is already

replicated in all servers in the old mapping and hence it cannot cause any data item migrations. Appropriately, if we were to illustrate m_{d_3}, we would always see that it has optimal response time of one.

Recall that the migration cost of the replicated declustering obtained at the end of phase two is the weight of the matching computed by the minimum weighted bipartite matching algorithm. Starting with this initial migration cost, proposed algorithms in this section try to reduce the overall migration and query processing costs.

The tuning between the objectives of minimizing migration and query processing costs can be performed by adjusting the frequency of migration queries. For example, one might argue that a migration query must have a frequency that is equal to at least two, since each migration requires a disk read from the source disk and a disk write at the destination disk.

After representing the migration of each data item as a separate query and setting the frequency of such queries appropriately, the number of objectives we try to optimize reduce from two to one, namely the minimization of query processing costs. This abstraction scheme also enables the utilization of a successful multi-way replicated refinement scheme proposed in Chapter 2. Via this multi-way replicated refinement scheme, we further improve the K-way replicated declustering obtained in the first two phases. The only changes we make in the multi-way refinement algorithms presented in Chapter 2 are slight modifications to the initial gain calculations and actual gain computations.

In the K-way refinement, for a migration query, if a move or a replication operation increases the number of data items requested from a server from one to two, then the gain of that move or replication operation is reduced by the cost of a data item migration. Similarly, for a migration query, if a move operation decreases the number of data items requested from the server that the data item was residing in from two to one, then the gain of that move operation is increased by the cost of a data item migration. Note that a replication operation can not decrease the migration cost.

3.3 Experimental Results

In our comparisons we used the same datasets described in Chapter 2. While testing the performance of the proposed query-log and mapping aware selective re-declustering scheme (re-SRD), the query sets for all datasets are divided into two equal parts. The first half is used for finding an initial replicated declustering of data items to disks using the SRA scheme proposed in Chapter 2. This mapping is used as the old (initial) data-item-to-disk mapping. In the experiments assuming query changes, the second half of the query sets are used as the new query set. In the experiments assuming disk addition or removal, on the other hand, query pattern changes are not assumed and the re-declustering problem for the same query set with increased or decreased number of disks is addressed.

All of the algorithms used in the experiments are implemented in C programming language, and experiments are conducted on a 2GHz Intel Core Duo machine with 2MB L2 cache and 2GB DDR2 667 MHz memory.

Query processing and migration requirement performances of the re-SRD scheme for the case where query pattern changes are observed is compared with the SRA algorithm on K=16, 24, 32 disks and the allowed overall replication ratio is varied from 10% to 100%. With 9 different datasets, 3 different disk counts, and 10 different replication ratio values, we present the results of 270 different experiment instances. For each SRA and re-SRD experiment instance, we report the average of 10 runs, since both algorithms use randomly generated initial feasible two-way declusterings in their replicated declustering phase.

Query processing and migration requirement performances of the re-SRD scheme for the case where disk addition and disk removal is performed is also compared with running the SRA algorithm from scratch for each new disk setting. The weighted bipartite matching scheme proposed in Section 3.2.2 is also utilized to match the newly obtained replicated declustering solution of SRA with the old mapping so as to minimize the migration costs of scratch-remap SRA scheme. We call this migration optimized SRA scheme mo-SRA.

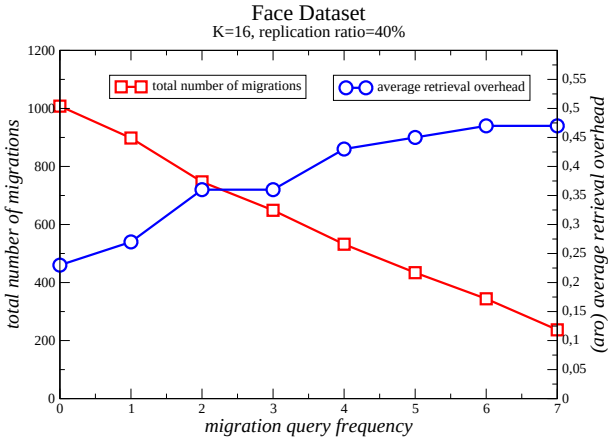


Figure 3.3: Average retrieval overhead and migration cost performances of re-SRD with changing migration query frequencies. The figures are for the query pattern change experiments on the *Face* dataset, $K = 16$, and replication ratio 40%.

The query processing performance of a given algorithm is evaluated in terms of the total number of migrations necessary to realize the mapping generated by the algorithm and the average retrieval overhead per query induced by the resulting replicated declustering.

Our first set of experiments evaluates the efficiency of the proposed query-log and mapping aware selective re-declustering scheme (re-SRD) in tuning between migration and query processing costs. Fig. 3.3 displays the change we observe in query processing overheads and total number of migrations with changing migration query frequencies for the Face dataset, $K = 16$, and replication ratio 40%. As seen in the figure, when we increase the frequency of the pseudo-queries added for representing migrations, the total number of migrations decrease significantly, whereas the aro values increase slightly. We should note here that similar results are obtained with the other datasets especially for replication ratios of 40% or lower. These results clearly show that the pseudo-query frequency parameter of re-SRA can successfully be used as a knob to tune the two objectives of the re-declustering problem. Depending on the requirements of applications, database admins can determine the value of migration query frequencies, e.g., high when migration is very costly and low when an increase in query processing costs is

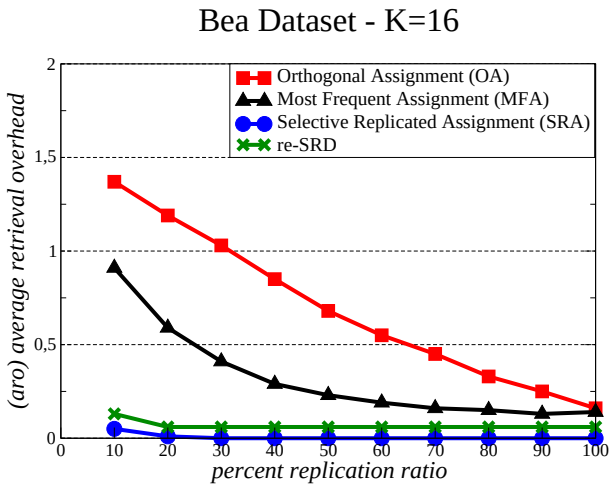


Figure 3.4: Average retrieval overhead performance of the replicated declustering schemes with respect to replication ratio. The figures are for the query pattern change experiments on the *Bea* dataset for $K = 16$.

unacceptable.

In the remaining of our experiments we fixed the pseudo-query frequencies to three with the assumption that migration of a data item is equivalent to three disk read operations. This is based on the idea that a migration, whether it be a move or replication, requires one disk read, one network transfer and one disk write operation, and these three operations has to be performed sequentially.

In Fig. 3.4, the performance of re-SRD is compared with the SRA and two other replicated declustering schemes, namely OA [15, 20] and MFA [34], in reducing average retrieval overheads over the *bea* dataset for $K = 16$. The OA and MFA schemes are selected since they are known to perform good for arbitrary queries. As seen in the figure, re-SRD performs slightly worse than SRA in reducing the average retrieval overheads, however, it performs significantly better than OA and MFA. This is expected since the sole objective of SRA is reducing average retrieval overheads whereas re-SRD also considers migration cost minimization. We should note that the results for other datasets display quite similar results and thus are not displayed here due to space considerations.

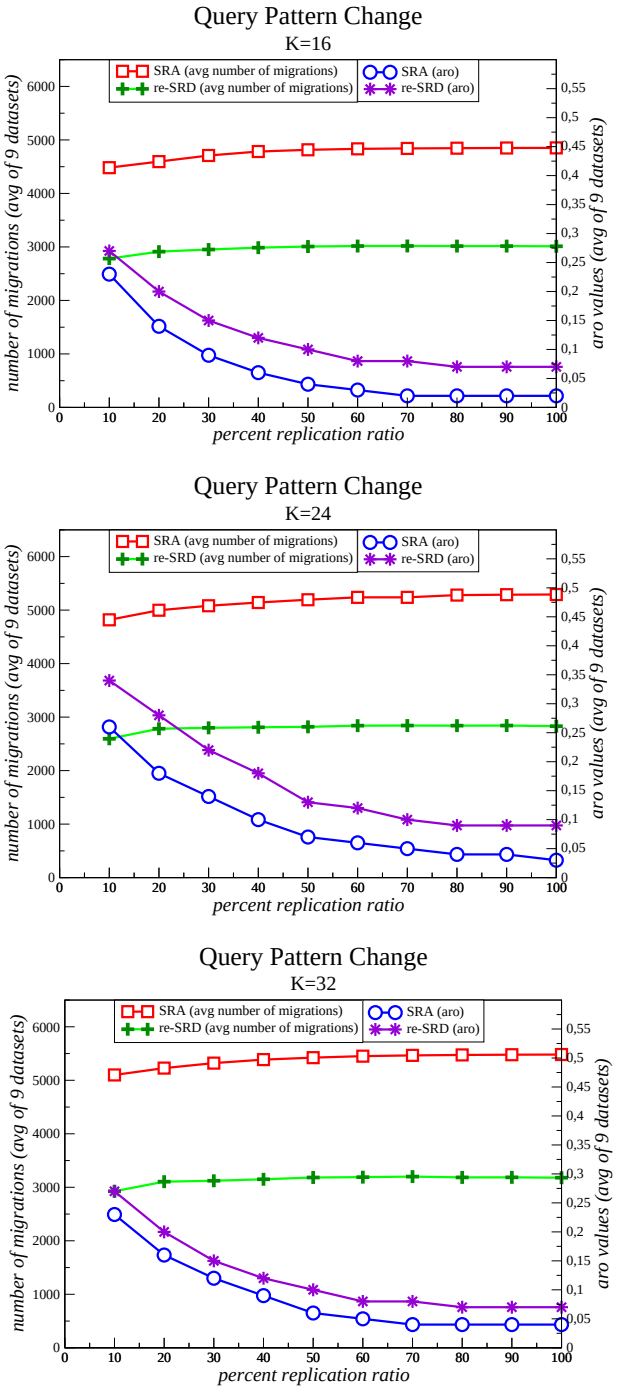


Figure 3.5: Average retrieval overhead and average number of migrations vs replication ratio figures for the datasets in the case of query pattern change experiments.

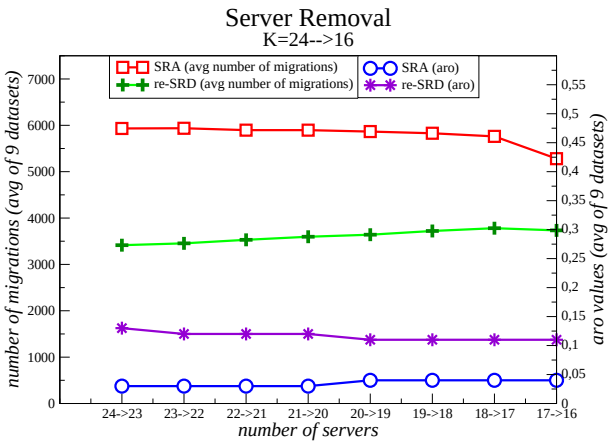


Figure 3.6: Average retrieval overhead and average number of migrations vs replication ratio figures for the datasets in the case of server removal experiments.

In Fig. 3.5, the performance of re-SRD is compared with SRA in terms of their performance in reducing average retrieval overheads and migration costs in the case of query pattern changes for $K = 16, 24, 32$ servers. Both the *aro* values and the number of migrations are averaged over the nine datasets. As seen in the figure, re-SRD *aro* results are slightly worse than that of SRA. However, re-SRD achieves significant reductions in migration costs. On average, re-SRD performs 43% less migrations than SRA. As also seen from the figure, with increasing replication ratio and increasing server counts, the total number of migrations slightly increase as expected.

A thorough analysis of Fig. 3.5 reveals that the effect of replication ratio to the performances of re-SRD and SRA are almost the same and changing the replication ratio does not change the relative performance of re-SRD and SRA. In the further experiments on server addition and removal, we fix the replication ratio parameter to 50% for the clarity of presentations. We should note that changing this parameter does not change the arguments we make in the following discussions.

Fig. 3.6 illustrates the *aro* and number of migration performances of SRA and re-SRD schemes for the case where server removals from the parallel database is being performed. In this experiment we assume that the number of servers are

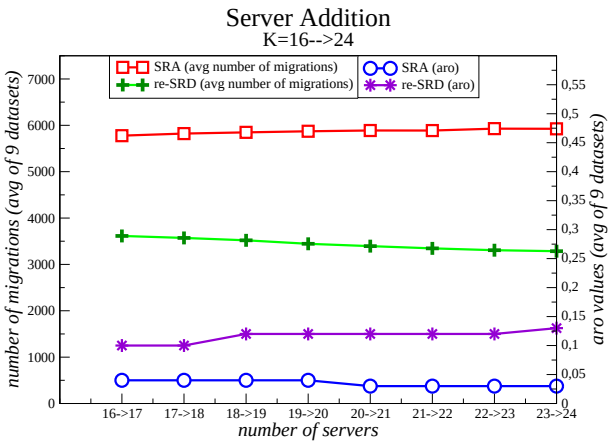


Figure 3.7: Average retrieval overhead and average number of migrations vs replication ratio figures for the datasets in the case of server addition experiments.

decreased from 24 to 16 via single server removals. After each removal, re-SRD generates a mapping respecting the previous data item assignments, whereas the mapping of SRA is constructed from scratch discarding the previous item mappings. As seen in the figure, the total number of migrations in re-SRA is significantly lower than that of SRA. As expected, SRA achieves lower *aro* values.

Fig. 3.7 illustrates the *aro* and number of migration performances of SRA and re-SRD schemes for the case where server additions to the parallel disk system is being performed. In this experiment we assume that the number of servers are increasing from 16 to 24 via single server additions. After each server addition, re-SRD generates a mapping respecting the previous data item assignments, whereas the mapping of SRA is constructed from scratch discarding the previous item mappings. As seen in the figure, the total number of migrations in re-SRA is significantly lower than that of SRA. As expected, SRA achieves lower *aro* values.

3.4 Related Work

Most of the existing re-declustering literature is about data availability issues and imbalanced load prevention schemes in case of failures. There are a few works

that tries to address changes in access frequencies as well.

One of the earliest declustering approaches is chained declustering [36], a scheme that assumes the servers are aligned over a ring and places replicas of data items onto consecutive servers over the ring. Chained declustering provides high availability and good load balance in the event of disk failures (if these failures do not occur on adjacent nodes). A more recent study ([37]) that stems from chained declustering and suggests which nodes can be closed/removed on purpose in a chain-declustered system for energy efficiency can be considered relevant to our discussions. Unfortunately, these approaches do not utilize query logs and provide a fixed assignment of data items to servers. Thus, they do not propose new data assignments under changing query workloads or addition/removal of disks.

In [38], an approach that tries to provide scalability for hash-based distributed files is discussed. Dynamic addition and removal of data items (records in a file) and changing query patterns are considered. The scalability issues are addressed via bucket (data item group) splits and migrations. Automatic server addition schemes to maintain fixed response times are also discussed, However, the approaches in [38] do not consider replication.

An automized data migration tool called Aqueduct that reacts to changing data access patterns by migrating "hot" data items is described in [39]. The Aqueduct system features a feedback control loop that regulates the speed of the data migration in the parallel disk system while hiding performance impact on the system. This approach is similar to our approach in the sense that it tries to ensure that the migration operations do not have a negative impact on query processing performance. However, unlike our approach which tries to strike a balance between migration and query processing costs, the schema in [39] tries to achieve this by performing migration only when all I/O requests are served. Another paper that uses replication and migration for improving throughput and load balancing is [40]. This approach, which is called DORA, assigns data items dynamically and integrates replication of files into the assignment to effectively distribute requests on "hot" data items across all servers of the system. DORA

has schemes for dynamic replica removal for “cold” files. Both works listed above can adapt to query pattern changes via dynamic assignments, however, non of them consider server addition or removal operations to the parallel database system.

Chapter 4

Query Processing in Replicated Inverted Indexes

Due to recent advances in big-data processing and serving technologies, utilizing term-partitioned indexes in parallel query processing systems became a viable alternative. In term-partitioned inverted indexes, replication of inverted lists associated with the most frequent terms is employed to improve the performance of the query processing system. In this chapter, we adopt a recently proposed replicated-hypergraph-partitioning-based approach for generating replicated, term-partitioned indexes and evaluate the performance of this approach against state-of-the-art partitioning and replication schemes. We also discuss various scheduling schemes that are required when replication is involved. We investigate these schemes on a realistic parallel query processing system We provide extensive experimental analysis performed up to 32 processors to show that proposed schemes are superior to the state-of-the-art alternatives.

4.1 Introduction

In this chapter, we present our approaches in implementing a parallel query processing system that supports term-based distribution and replication. Even

though search systems are used extensively, and the internal working structures of such systems are more or less known generally, we found that algorithmic and engineering-wise decision made during the implementation of such systems greatly effect the overall performance.

In state-of-the-art search engine systems, document-based inverted index distribution is preferred over the term-based distribution due to the following two reasons: (i) following a parallel crawling phase, building a document-based distributed index becomes easier than building a term-based distributed index, (ii) document-based distributed indexes achieve better load-balancing performance during query processing. However, due to recent advances in big-data processing technologies, it is possible to build term-based distributed inverted indexes in acceptable times [41]. Furthermore, the load-balancing inferiority of term-based distributed indexes can be alleviated via replication schemes. Thus, together with replication, term-based distribution becomes a viable alternative to doc-based distribution.

In this chapter, we consider a parallel query processing system utilizing a term-partitioned inverted index where replication of terms are applied to improve performance. Our contributions are fourfold:

- We implement a successful parallel query processing system and we provide details of our implementation and the reasons behind our design choices.
- We utilize a successful replication approach based on replicated hypergraph partitioning, which was recently proposed in [5]. Our experimental results demonstrate that this approach is much more successful than the state-of-the-art partitioning and replication schemes.
- When there is replication, the problem of selecting the replica to be used in query processing arises. We show that this problem can be reduced to the set-cover problem. Furthermore, we propose various heuristics for scheduling replicated terms.
- We provide extensive experimental studies performed up to 32 processors

on our parallel query processing system and report performance analysis with respect to various different metrics.

This chapter is organized as follows: In Section 4.2, we provide the necessary background. In Section 4.3, we describe the details of our parallel query processing system. In Section 4.4, we investigate the performance of various existing partitioning and replication schemes. In Section 4.5, we discuss scheduling issues in replicated term-partitioned indexes. In Section 4.6, we compare investigated methods and algorithms over the proposed parallel query processing system and discuss the results. Finally, in Section 4.7, we briefly review the related literature.

4.2 Background

4.2.1 Basics of Query Processing

The main objective of query processing is to find out the relevant documents to a user query $q = \{t_1, t_2, \ldots, t_n\}$ and display them to the user. A set of relevant documents $\{d_1, d_2, \ldots, d_s\}$ for q is returned back to the user according to the result of various similarity calculations [42] between the documents in the collection and the query, where the found relevant documents are sorted in non-increasing order with respect to used similarity measure. We use the widely accepted tf-idf weighting scheme together with the vector space model [43] for similarity calculations. However, any other similarity measure can easily be integrated into our system.

Since it is not practical to make the similarity calculations directly using the raw document contents, documents are first converted into an inverted index [44, 45], $\mathcal{L} = \{(t_1, \mathcal{I}_1), (t_2, \mathcal{I}_2), \ldots, (t_{|\mathcal{T}|}, \mathcal{I}_{|\mathcal{T}|})\}$. In the inverted index data structure, each term t_i in the vocabulary of the collection has an associated inverted list $\mathcal{I}_i$, which contains a set of postings $\mathcal{I}_i = \{p_1, p_2, \ldots, p_n\}$. Each posting p_j of the inverted list associated with term t_i is a tuple (d_j, w_j), where d_j is the id of a document that contains t_i and w_j [46] is the relevance between the document d_j

and the term t_i.

There are several phases (possibly interleaving) of query processing: *creation, update, extraction, selection*, and *sorting* [47]. After a query is submitted for processing, an accumulator is *created* to store the similarity scores for the documents. An entry in an accumulator array A is notated with $a_j = (d_j, s_j)$, where s_j is reserved for storing the final similarity score of d_j with respect to used similarity measure. During *update* phase, inverted lists corresponding to the terms in the query are fetched from the disk and the entries in the accumulator are updated accordingly. The update of these entries changes with respect to used document matching logic, which can be `AND` (conjunctive mode) or `OR` (disjunctive mode). In `AND` logic, only the documents that include all query terms are matched, whereas in `OR` logic, the documents that include at least one of the query terms are matched. The score s_j for a_j is simply computed by adding w_j values of the corresponding postings for d_j in the inverted lists of the terms of the query:

$$s_j = \sum_{t_k \in q_i \wedge p_j \in \mathcal{I}_k} w_j.$$

Then, the nonzero entries in the accumulator are *extracted* and the documents with the top s scores are *selected*. Finally, the selected documents are *sorted* in non-increasing order of their similarity scores (s_j) and returned back to the user in this order. The reader is referred to [47] for an extensive analysis of alternative sequential query processing implementations.

4.2.2 Term-Based Parallel Query Processing

In term-based index distribution, the inverted index $\mathcal{L}$ is partitioned into K pairwise disjoint subsets $\mathcal{L}_1, \ldots, \mathcal{L}_K$, where K is the number of index servers in the parallel query processing system. Formally,

$$\mathcal{L} = \bigcup_{k=1}^{K} \mathcal{L}_k, \ \mathcal{L}_i \cap \mathcal{L}_j = \emptyset \ \text{ for } 1 \leq i < j \leq K. \tag{4.1}$$

Each index server IS_k is responsible for maintaining the sub-index $\mathcal{L}_k$, where

$$\mathcal{L}_k = \{(t_i, \mathcal{I}_i) : t_i \text{ is assigned to } IS_k\}.$$

Table 4.1: Notations used in this chapter.

Symbol	Description
$\mathcal{T}$	Terms in the collection, the lexicon
$\mathcal{D}$	Documents in the collection
K	Number of index servers
IS_k	k^{th} index server
t_i	A term in the collection
d_j	A document in the collection
$\mathcal{L}$	The set of inverted lists
$\mathcal{L}_k$	The terms and their inverted lists associated with IS_k
$\mathcal{I}_i$	The inverted list associated with t_i
p_j	A posting in an inverted list
(d_j, w_j)	The document and the weight associated with p_j
q or q_i	A user query
q_i^k	The subquery generated for IS_k
P_i^k	The partial answer set for q_i constructed by IS_k
A	Accumulator array
a_j	An entry in the accumulator
(d_j, s_j)	The document and the score associated with a_j
O_r	The queue of receptionist
Q_k	The queue of IS_k
r	Maximum partial answer set size for an index server
s	The number of documents to be returned to the user

The assignment of terms and their inverted lists to index servers can be performed in various ways. This process has a crucial effect on the performance of the parallel system and must be done carefully. We discuss this issue in great detail in Section 4.4.

A common technique used in improving performance of the parallel query processing systems is replication of terms and their inverted lists. When replication of terms and their inverted lists are allowed, a term and its inverted list can be assigned to more than one index server. Thus, the constraint $\mathcal{L}_i \cap \mathcal{L}_j = \emptyset$ for $1 \leq i < j \leq K$ in Equation 4.1 is no more valid. Fig. 4.1 shows an example of term-based index partitioning with replication included. Hereafter, whenever we use term replication, we actually mean the replication of that term and its inverted list.

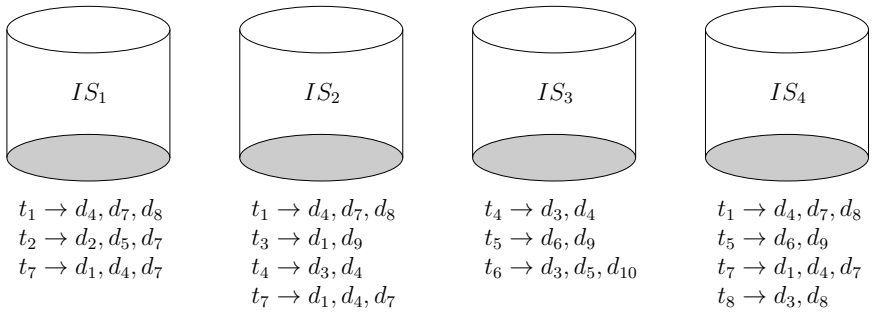


Figure 4.1: An example of term-based partitioning for $\mathcal{T} = \{t_1, \ldots, t_8\}$ ($|\mathcal{T}| = 8$), $\mathcal{D} = \{d_1, \ldots, d_{10}\}$ ($|\mathcal{D}| = 10$) on four index servers ($K = 4$). There are four replicated terms, t_1, t_4, t_5, and t_7. Each index server has a local sub-index that consists of terms and inverted lists that it is assigned to. For example, for IS_2, we have $\mathcal{L}_2 = \{(t_1, \mathcal{I}_1), (t_3, \mathcal{I}_3), (t_4, \mathcal{I}_4), (t_7, \mathcal{I}_7)\}$. A term and its inverted list are given in the form of $t_i \rightarrow \mathcal{I}_i$. The weight values of the documents in the postings are omitted for clarity.

4.3 Parallel Query Processing

Our target parallel query processing system is a shared-nothing parallel architecture with K index servers and a single central receptionist, where the central receptionist and each index server are running on separate nodes. Interprocessor communication and coordination are achieved via explicit message passing. A PC cluster forms a typical case of our target architecture. In this setting, we focus on the central broker parallel query processing scheme, which is by large the standard coordination approach utilized in parallel search systems.

The central broker (CB) parallel query processing scheme is a master-slave type of architecture. In a typical CB scheme, there is a single receptionist (master), which collects the incoming user queries and forms subqueries from them to be sent to the corresponding index servers (slaves). The index servers are responsible for generating partial answer sets to the received subqueries using their local inverted indices. The generated partial answer sets are later merged/updated into a global answer set at the receptionist, forming the answer set for the query, which is then sent back to the user. In this section we propose algorithms that run on the receptionist and the index servers for the CB scheme and give implementation level details for these algorithms.

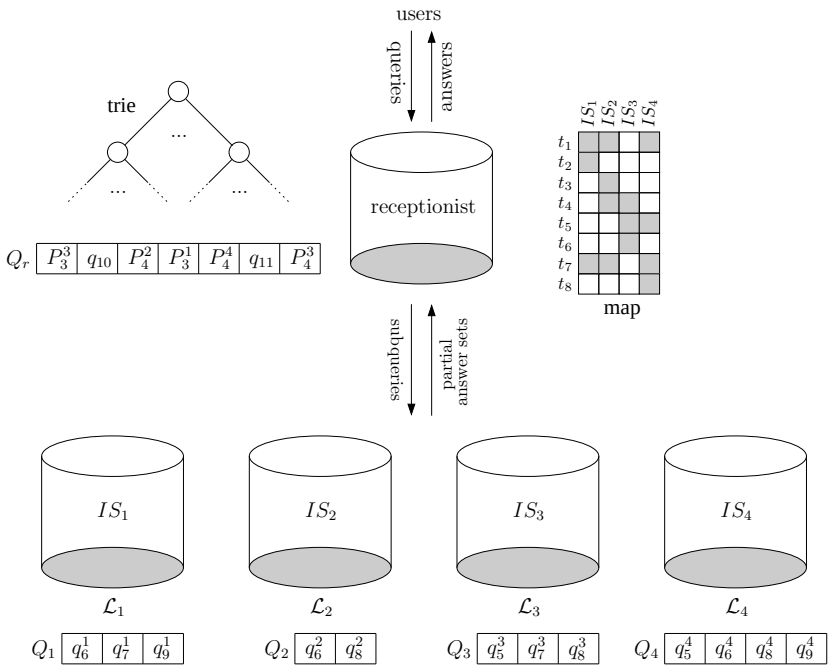


Figure 4.2: A snapshot of a parallel query processing system in CB scheme. There are four index servers and a single receptionist. The index servers' local indices ($\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4\}$) correspond to the term-based partitioning given in Fig. 4.1. There are 11 queries submitted to the receptionist. Two of them (q_1 and q_2) are already answered and thus not in the system, two of the submitted queries' (q_3 and q_4) PASs are in Q_r, five of them (q_5, q_6, q_7, q_8 and q_9) are being processed at the index servers and two of them (q_{10} and q_{11}) have just been received from users which are in Q_r.

Before starting to process user queries, the receptionist and the index servers allocate and initialize necessary data structures. In this phase the receptionist creates a *trie* (aka radix tree or prefix tree), in which it keeps the terms in the collection and their associated ids. Upon receiving a query, the id of a query term is accessed from this trie in $O(\ell)$ memory accesses where ℓ is the length of that term. The receptionist also contains a term-to-index server map (simply referred to as map), which contains information about the assignment of terms to index servers in a term-based partitioning. The receptionist and the index servers use a queue while processing user queries. The queue maintained by the receptionist (Q_r) contains queries received from the users and the partial answer sets (PASs) received from the index servers, whereas the queue maintained by each index server (Q_k at IS_k) contains only the subqueries received from the receptionist.

Fig. 4.2 shows a snapshot of a parallel query processing system in central broker architecture that uses the term partitioning given in Fig. 4.1. A subquery of a query q_i that is constructed for IS_k is denoted as q_i^k and the partial answer set (PAS) generated for this query by IS_k is denoted as P_i^k. In the figure, the receptionist maintains three basic structures, a trie, a map and a queue that contains queries received from users as well as partial answer sets received from index servers. An entry in the map at the receptionist indicates whether IS_k stores t_i. For example, t_4 is stored at two index servers IS_2 and IS_3 (see Fig. 4.1). In the map at Fig. 4.2, these corresponding entries are displayed as shaded. As mentioned, the queues at index servers contain only subqueries received from receptionist.

4.3.1 Index Server Algorithm

Our index server algorithm (Algorithm 9) consists of a single infinite while loop (lines 1–18) in which the index server

1. periodically probes for incoming subqueries,
2. receives incoming subqueries,
3. produces PASs for subqueries,
4. and sends the generated PASs to the receptionist.

As the first action of its loop, IS_k probes for incoming subqueries from the receptionist and if the probe value is true, receives the incoming subquery and enqueues it to its queue Q_k (lines 3–5). When a subquery is dequeued from Q_k, IS_k processes the postings in the inverted lists of the subquery terms and updates the scores of documents in its accumulator array (A). The update of the accumulator array (lines 9–11) changes with respect to used document matching logic.

As an example consider a subquery $q_i^1 = \{t_1, t_7\}$ received by IS_1 for the term-based partitioning given in Fig. 4.1. In `AND` logic, the partial answer set

Algorithm 9: CB algorithm running on Index Server IS_k.

```
   Input: matchingLogic, r
 1 while true do
 2     PROBE whether a subquery is received from the receptionist
 3     if PROBE = true then
 4         Receive subquery q_i^k
 5         ENQUEUE(Q_k,q_i^k)
 6     if Q_k ≠ ∅ then
 7         q_i^k ← DEQUEUE(Q_k)
 8         A ← ∅  ▷ Initialize the accumulator array
 9         foreach t_i ∈ q_i^k do
10             foreach p_j = (d_j, w_j) ∈ I_i do
11                 Update the a_j = (d_j, s_j) entry in A with respect to given matching logic
12         if matchingLogic = AND then
13             P_i^k ← SELECT all nonzero entries in A
14         else if matchingLogic = OR then
15             P_i^k ← SELECT nonzero entries with top r scores in A
16         SORT P_i^k with respect to document ids
17         WAIT for previous send to finish
18         SEND P_i^k to the receptionist
```

P_i^1 generated by IS_1 will include the common documents in the inverted lists $t_1 \rightarrow \{d_4, d_7, d_8\}$ and $t_7 \rightarrow \{d_1, d_4, d_7\}$ which are d_4 and d_7, and the summation of the weight values of these documents in the postings of the corresponding subquery terms. In `OR` logic, the partial answer set will include the documents in the union of these inverted lists which are d_1, d_4, d_7 and d_8, and the summation of the weight values of these documents in the postings of the corresponding subquery terms. The update of entries in the accumulator array (lines 9–11) is performed in such a way that after the update is complete, if the document matching logic is `AND`, the non-zero scoring documents are the ones which are in the *intersection* of the inverted lists of the subquery terms, and if the document matching logic is `OR`, the non-zero scoring documents are the ones which are in the *union* of the inverted lists of the subquery terms.

For a query, the update of document scores begins at the index servers and finishes at the receptionist since an entry $a_j = (d_j, s_j) \in A$ may be existent in two or more PASs for q_i that are sent by different index servers. In this case, the s_j values need to be summed up at the receptionist to compute the final score for d_j. Due to this observation, the index servers sort the entries of the PASs using document ids as keys. This design choice is made so as to reduce the bottleneck at the receptionist by reducing its merge-and-update cost, at the expense of increasing the computational load at the index servers. Long accumulator arrays

sent from index servers to the receptionist induce high communication volume as well as high computational cost at the receptionist due to the merge-and-update operations.

To alleviate this problem for `OR` document matching logic, we adopt a slight modification of the accumulator limiting approaches mentioned in [48], where we restrict our PAS size to r by selecting top r documents with respect to their scores in each index server. As mentioned in the literature, this restriction does not degrade the query processing quality. Note that for `AND` document matching logic, the size limitation of PASs can have a great negative impact on the correctness and the quality of the returned answers, thus we do not limit the size of PASs in `AND` logic. The reasons behind this phenomenon are explained later in Section 4.3.2 in receptionist algorithm.

Returning to Algorithm 9, if document matching logic is `AND` (lines 12–13), all non-zero entries in the accumulator array are selected for the PAS P_i^k. If document matching logic is `OR` (lines 14–15), the nonzero entries with top r scores are selected for P_i^k in linear time. Then the index server sorts P_i^k (line 16) with respect to document id fields of the entries, enabling the receptionist to merge-and-update PASs efficiently. Finally, the index server waits for previous send operations to finish and then sends P_i^k to the receptionist (lines 17–18).

4.3.2 Receptionist Algorithm

Our receptionist algorithm (Algorithm 10) also contains an infinite while loop (lines 1–32). Within the infinite loop, the receptionist

1. continuously checks if any queries or PASs are received,
2. forms subqueries from queries and sends them to index servers,
3. merges and updates PASs,
4. and displays the final answer sets to users.

Algorithm 10: CB algorithm running on the receptionist.

```
   Input: matchingLogic, s
 1 while true do
 2     TEST whether any queries are received from clients
 3     if TEST = true then
 4         foreach received query q do
 5             ENQUEUE(Q_r,q)
 6     foreach index server IS_k do
 7         TEST whether a P_i^k is received from IS_k
 8         if TEST = true then
 9             ENQUEUE(Q_r,P_i^k)
10     if Q_r ≠ ∅ then
11         x ← DEQUEUE(Q_r)
12         if x.type = query then
13             Let the dequeued query be q_i
14             foreach IS_k do
15                 q_i^k ← ∅
16             qmap ← SCHEDULE(q_i,map)
17             foreach term t ∈ q_i do
18                 k ← qmap[t]
19                 q_i^k ← q_i^k ∪ {t}
20             foreach IS_k do
21                 if q_i^k ≠ ∅ then
22                     SEND subquery q_i^k to IS_k
23         else if x.type = PAS then
24             Let the dequeued PAS P_i^k belong to q_i sent by IS_k
25             if matchingLogic = AND then
26                 A[i] ← INTERSECT and UPDATE A[i] with P_i^k
27             else if matchingLogic = OR then
28                 A[i] ← COMBINE and UPDATE A[i] with P_i^k
29             if P_i^k is the last PAS for q_i then
30                 A[i] ← SELECT entries in A[i] with top s scores
31                 SORT A[i] with respect to score fields
32                 DISPLAY A[i] to client
```

As the first action within this while loop, the receptionist checks for incoming queries from users, and if any queries have been received, it enqueues them to its queue Q_r (lines 2–5). Similarly, it checks for incoming PASs from each index server, and if any PASs have been received, it enqueues them to the same queue (lines 6–9).

The receptionist dequeues an item from its queue (line 11) and takes different actions depending on the type of the item dequeued. If the dequeued item is a query (lines 12–22), the receptionist first parses the query terms in order to find their ids using the trie. It then forms subqueries and sends them to the corresponding index servers (lines 14–22). The subquery forming process includes another procedure called SCHEDULE, which basically, given a query and a map, returns the set of index servers that will process the given query.

As an example, consider $q_5 = \{t_1, t_5, t_7\}$ whose subqueries are already formed and distributed as shown in Fig. 4.2. Supposed that the call to SCHEDULE(q_5,map) returned $\{IS_4, IS_3, IS_4\}$ which is assigned to variable qmap. This means t_1 will be processed on IS_4, t_5 will be processed on IS_3, and t_7 will be processed on IS_4. With this information, the subqueries are formed as $q_5^1 = \emptyset$, $q_5^2 = \emptyset$, $q_5^3 = \{t_5\}$, and $q_5^4 = \{t_1, t_7\}$. The non-empty subqueries q_5^3 and q_5^4 are then sent to IS_3 and IS_4, respectively. Note that since replication is involved in term-based partitioning, there are possibly multiple ways to form subqueries. In the example of scheduling terms of q_5, all query terms t_1, t_5, and t_7 could be scheduled to IS_4, or in an alternative scheduling t_1 could be scheduled to IS_1, t_5 could be scheduled to IS_3, and t_7 could be scheduled to IS_2. We investigate different scheduling heuristics in Section 4.5.

If the dequeued item is a PAS (lines 23–32), the receptionist first retrieves the id of the query that this PAS belongs to (line 24). We adopt a two-way merge-and-update algorithm for forming an answer set from received PASs for q_i. In this approach, if the received PAS is the first one for q_i, then it becomes the initial accumulator array (A) for that query. Otherwise, the received PAS is merged-and-updated with the existing accumulator array immediately after it has been received. Clearly, before the last two-way merge, not all the scores are complete and the extraction and selection operations cannot be initiated for forming the final answer set that will be sent back to the user.

The two-way merge-and-update algorithm can be considered as an extension of the merge operation used in mergesort algorithm, which merges two sorted sub-lists into a sorted list in linear time. Like the merge operation, the algorithm advances two pointers over two document-id sorted accumulator arrays. In both `AND` and `OR` logic, if the two pointed entries' document ids are the same, their scores are added and stored as a single entry in the resultant array. If they are not the same, in `AND` logic, since only common documents must be matched, the entry with the small document id is discarded, whereas in `OR` logic, since we take the union of the documents, the score value of the entry with smaller document id is directly stored in the resultant array.

For example, consider the query $q_4 = \{t_1, t_4, t_7\}$ whose all PASs are already in the queue of the receptionist in Fig. 4.2. Assume t_1 is scheduled to be processed at IS_2, t_4 is scheduled to be processed at IS_3, and t_7 is scheduled to be processed at IS_4. Given this information, the PAS contents regarding q_4 in Q_r are $P_4^2 = \{d_4, d_7, d_8\}$, $P_4^3 = \{d_3, d_4\}$, and $P_4^4 = \{d_1, d_4, d_7\}$. The first PAS for q_4 in Q_r is P_4^2, thus the accumulator array for q_4 at the receptionist will be initialized to $\{d_4, d_7, d_8\}$. The next PAS to be processed for q_4 is P_4^4, and the last one is P_4^3. For `AND` logic, only the common documents will form the next accumulator array, thus after processing P_4^4 the accumulator array will include $\{d_4, d_7\}$, and after processing P_4^3 the accumulator array will only include $\{d_4\}$. For `OR` logic, union of the PASs will form the next accumulator array, thus after processing P_4^4 the accumulator array will include $\{d_1, d_4, d_7, d_8\}$, and after processing P_4^3 the accumulator array will include $\{d_1, d_3, d_4, d_7, d_8\}$. Note that in the update of common documents, the score values of these documents are summed (they are omitted for clarity). This example clearly illustrates why index servers need to send all of their computed PASs in `AND` logic: the receptionist is not only responsible for adding scores of the documents (as in `OR` logic), but also for selecting only the common documents that exist in all PASs of q_i.

The algorithm proceeds with merging-and-updating the dequeued PAS with the accumulator array with respect to document matching logic (lines 25–28) as mentioned above. If this PAS is the last PAS for q_i, it means the answer set is ready to be sent back to the user. In this case, the receptionist selects the documents with top s scores in the accumulator array, sorts these documents with respect to score values, and then displays them to the user (lines 29–32).

4.4 Analysis of Index Partitioning and Term Replication Schemes

In this section, we explain commonly used methods for achieving partitioning and replication in parallel query processing systems as well as recently proposed,

novel hypergraph-partitioning-based (HP-based) methods for achieving partitioning and replicated partitioning [5] of inverted indexes.

4.4.1 Bin-packing-based Index Partitioning

In the bin-packing-based index partitioning scheme, each term is associated with a certain weight such as frequency of that term in the query log or its inverted list length. Assigning a term to an index server corresponds to assigning the load associated with that term to that index server. Thus, obtaining a balance on the cumulative weights of the terms assigned to index servers corresponds to balancing the loads of these index servers. Obtaining a term-partitioned inverted index while minimizing the load (e.g. storage, computational, etc.) imbalance among the index servers can be reduced to the minimum makespan scheduling problem [49] as mentioned in [50] if the index servers are identical. In this reduction, the terms correspond to tasks, the weights of terms correspond to task lengths, and index servers correspond to processors. In this way, minimizing the makespan corresponds to minimizing the load imbalance in term-partitioned indexes.

Minimum makespan scheduling problem is known to be NP-hard [51], and thus, is generally addressed with heuristics. In parallel information retrieval systems, a best-fit decreasing heuristic that is generally used in bin-packing problem (hence the name) is reported to give good load balance values [52, 53] for obtaining term-partitioned indexes. This heuristic is also used in [50] to allocate documents to servers.

In the bin-packing heuristic for term-partitioned indexes, the weight associated with each term is generally either the inverted list length of that term, or the number of queries containing this term if the query log is utilized [52, 53]. If inverted list lengths are used, then this is equivalent to balancing the storage load of the index servers. However, if query term frequencies are used, we actually try to balance the number of term accesses performed in each index server. Note that if query term frequencies are used, there may remain several terms that need to

be assigned to index servers which do not occur in the query log. To distribute such unqueried terms, it is possible to follow the same bin-packing scheme given in [53] that utilizes inverted list lengths of terms.

4.4.2 Most Frequent Term Replication

A common and widely used approach for replication in parallel query processing systems is the replication of most frequent terms [52, 53]. In this scheme, a certain amount of most frequent terms (with their inverted lists) are replicated in each index server. As mentioned in Section 4.4.1, the frequency of a term can be interpreted as either its inverted list length or the number of queries containing this term if query log is utilized. Generally, the terms having longer inverted lists or more query term appearances have higher priorities for replication. Note that the most frequent term replication is independent of the underlying partitioning scheme.

The motivation behind the most frequent term replication is that the replication of high frequency terms, which are the most probable causes of load imbalance, is likely to improve the overall performance of the system in terms of average response time and query throughput. This is because by replicating such terms, we prevent bottlenecks in the system via distributing the loads of the highly accessed terms evenly among all index servers. Thus, the most frequent term replication improves the performance of the parallel query processing system via using replication as a load balancing tool.

4.4.3 Hypergraph-partitioning-based Index Distribution

A recently proposed, efficient index distribution scheme is HP-based index distribution. In this scheme, a hypergraph is constructed by utilizing the information in past query logs. Then, this hypergraph is partitioned for the given number of index servers to obtain an index distribution. Formally, given a query log consisting of m queries $q_1, \ldots, q_m$, and the terms that appear in each query $q_j = \{t_1, \ldots, t_r\}$

for $j = 1, \ldots, m$, a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is constructed where the queries correspond to the nets and the terms (with their inverted lists) correspond to the vertices of the hypergraph. More specifically, a query $q_j = \{t_1, \ldots, t_r\}$ in the query log is modeled by a net n_j and the terms of this query are modeled by vertices $v_1, \ldots, v_r$, where the vertices connected by n_j correspond to terms that occur in q_j. Thus, if there are m queries and a total of n distinct terms that appear in the query log, the corresponding hypergraph will consist of m nets and n vertices. After partitioning $\mathcal{H}$ and obtaining a partition $\Pi = \{\mathcal{V}_1, \ldots, \mathcal{V}_K\}$, the part $\mathcal{V}_k$ is mapped to index server IS_k and each $v_i \in \mathcal{V}_k$ associated with t_i and its inverted list is stored in IS_k. From now on, when presenting hypergraphs in figures, we use t_i for representing vertices and q_j for representing nets.

Fig. 4.3 shows a four-way partition obtained using hypergraph partitioning for the hypergraph constructed from four queries and eight terms. After obtaining a four-way partition, the vertices (which correspond to terms) in obtained parts are assigned to the corresponding index servers which are illustrated in Fig. 4.3 as IS_1, IS_2, IS_3, and IS_4. The connectivity set of a net q_j indicates the index servers that will participate in answering this query. For example, the connectivity set of q_3 is $\{IS_2, IS_3, IS_4\}$. Thus, these index servers will participate in answering q_3.

In the hypergraph model we utilize, the weight of a vertex associated with t_i is assigned the number of queries that contain t_i in the query log. In other words, we use query term frequencies as vertex weights. Therefore, maintaining balance in hypergraph partitioning corresponds to balancing the number of term and inverted list accesses performed by each index server.

We assign unit costs to nets and use connectivity metric for cutsize computation to correctly capture the number of index servers involved in answering a query. In this model, the cutsize of a partition Π corresponds to the total number of index servers involved in answering the queries in the log. Thus, we try to minimize the total number of index servers that are involved in answering queries, which is especially useful for `AND` document matching logic. That is because if a query can be answered from a single index server in `AND` logic, the inverted lists

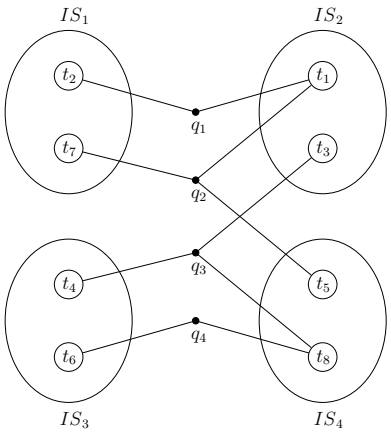


Figure 4.3: A four-way partitioning of the hypergraph constructed from queries $q_1 = \{t_1, t_2\}, q_2 = \{t_1, t_5, t_7\}, q_3 = \{t_3, t_4, t_8\}$, and $q_4 = \{t_6, t_8\}$.

will be `AND`ed (intersection) and the resulting PAS will be quite small which will in turn lead to a small communication volume. This is not valid for `OR` logic since when the inverted lists are `OR`ed (union) in an index server, the resulting PAS will be larger. However, this is not a problem in our implementation because we limit the PAS size in `OR` logic (Section 4.3.1). Thus, we can avoid large communication volumes in `OR` logic.

4.4.4 Replicated Hypergraph-partitioning-based Index Distribution

A recent study that addresses replication in hypergraphs is discussed in [5], and a tool named `rpPaToH` that is capable of replicating vertices of an undirected hypergraph to improve the target objective via utilizing a given amount of replication is proposed. In [5], generation of term-partitioned replicated inverted indexes is given as an example for the use of `rpPaToH`.

We apply the same approach used in [5] to replicate vertices in the hypergraph model introduced in 4.4.3, utilizing `rpPaToH`. In this way, replication can *directly* be used to improve the objective modeled by the hypergraph partitioning. This forms the main difference of our approach compared to the most frequent

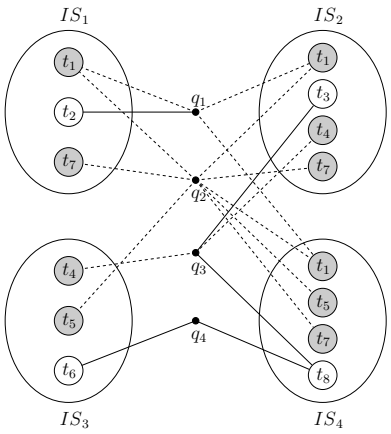


Figure 4.4: A four-way replicated partition of the hypergraph constructed from the same query set given in Fig. 4.3. The terms t_1, t_4, t_5, and t_7 are replicated. The replicated vertices are shown as shaded. The pins of the nets to the replicated vertices are illustrated as dashed lines.

term replication mentioned in Section 4.4.2. By using a hypergraph partitioning model and performing vertex replication during partitioning, we can actually address a *particular* objective while maintaining balance on a certain criterion (see Section 4.4.3 for the objective and balance constraint of the used hypergraph model in this work). Moreover, using replicated hypergraph partitioning allows us to replicate terms in a finer granularity compared to the most frequent term replication, which replicates frequent terms to all index servers. Instead of replicating frequent terms to all index servers greedily, we replicate terms to the index server(s) where they are most "needed" according to the defined objective. Note that, in index servers, the replication of terms and their inverted lists has only storage overhead and does not incur any additional computational cost neither in most frequent term replication nor in replicated hypergraph partitioning.

Fig. 4.4 shows a four-way replicated partition of the hypergraph given in Fig. 4.3. As seen from the figure, t_1 is replicated in IS_1, IS_2, and IS_4, t_4 is replicated in IS_2 and IS_3, t_5 is replicated in IS_3 and IS_4, and t_7 is replicated in IS_1, IS_2, and IS_4. Replicating terms in a parallel query processing system carries the scheduling problem within itself. When a replicated term is requested by a query, the central broker has to make a scheduling decision about which index server will be used to process that replicated term. For example, consider $q_1 = \{t_1, t_2\}$ in Fig. 4.4. The replicated term t_1 has three replicas and it can

be processed by IS_1, IS_2, or IS_4, whereas t_2 has to be processed in IS_1 since it is not replicated and has only one replica. Making this decision can affect the performance of the parallel query processing system significantly and thus must be handled very carefully. We investigate the scheduling problem in Section 4.5.

4.5 Investigated Query Scheduling Heuristics

When there are replicated terms, the problem of selecting which replicas to use arises. Depending on the selection made, the scheduled subqueries can change dramatically. Recall that to schedule queries, we maintain a map of term-to-index-server assignment at the central broker. As mentioned in Section 4.4.3, the investigated hypergraph model tries to minimize the number of index servers involved in answering a query. In this way the communication volume, which is one of the most important factors that determines the overall performance of a parallel query processing system, can be reduced. By using the scheduling flexibility provided by replication, we can also balance the load of the index servers. This section introduces algorithms to schedule replicated terms. The spectrum of algorithms presented in this section have both extremes. In one extreme we only consider minimizing the number of index servers involved in answering a query, and in the other extreme we only consider balancing the load of the index servers using dynamic information about them. We also propose a hybrid scheme that mediates these two extremes.

To see how influential scheduling decisions can be in minimizing the number of index servers involved in answering a query, consider q_2 in Fig. 4.4. When the central broker is to form subqueries for this query, it can select from several options. One of them is $q_2^1 = \{t_1\}, q_2^2 = \emptyset, q_2^3 = \{t_5\}$, and $q_2^4 = \{t_7\}$, which requires three index servers to answer this query. Another approach would be to schedule q_2 as $q_2^1 = \emptyset, q_2^2 = \emptyset, q_2^3 = \emptyset$, and $q_2^4 = \{t_1, t_5, t_7\}$, which schedules all terms to IS_4 and requires only one index server for answering q_2.

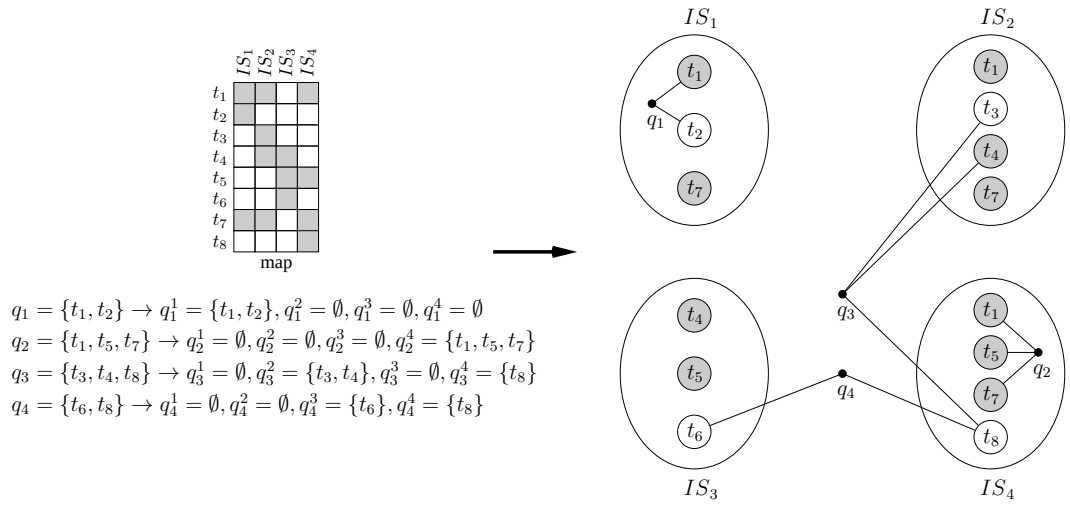


Figure 4.5: The schedule obtained after running the set-cover-based scheduling heuristic on the replicated partition and the queries given in Fig. 4.4. Note that after scheduling and selecting instances of the replicated terms for the queries, the dashed pins in Fig. 4.4 became normal pins.

4.5.1 Reduction to Set Cover Problem and Set-Cover-Based Scheduling

In this section, we show that minimizing number of index servers involved in answering a *single* query in replicated and term partitioned indexes can be reduced to the set cover problem. We follow the approach mentioned in [5] where the same problem is solved for the final cutsize computation in replicated partitioning for undirected hypergraphs.

Before showing how this reduction is done, we need to eliminate the non-replicated terms requested by a query since there exist only a single instance of a non-replicated term and it is obvious which index server is going to process it. An immediate observation following this proposition is that, since we have to use certain index servers for non-replicated terms of a given query, we can (and should) try to schedule the replicas of the replicated terms occurring in this query to these index servers, if possible. In this way, we do not increase the number of index servers involved in answering the given query after scheduling non-replicated terms while choosing replicas of the replicated terms.

For example, consider $q_1 = \{t_1, t_2\}$ in Fig. 4.4. The IS_1 has to involve in

answering this query since it is the only index server that stores t_2. However, there are three alternatives for t_1, which are IS_1, IS_2, and IS_4. The above-mentioned observation simply tells us to schedule t_1 to one of the index servers that are already used for processing non-replicated terms, which is IS_1 in this case. By doing so, we do not increase the number of index servers involved in answering this query while scheduling replicated terms of it. After eliminating non-replicated terms and replicated terms whose replicas we can schedule to the index servers we use for non-replicated terms, we are left only with the replicated terms which we cannot process at already chosen index servers. This case can be seen for q_2, all of whose terms are replicated and cannot be scheduled to the index servers chosen for non-replicated terms.

In such cases, for the remaining replicated terms, the scheduling problem reduces to the set cover problem which is known to be NP-complete [54]. Again, consider q_2 in Fig. 4.4. If we model this query and its unscheduled replicated terms with the set-cover problem, the ground set for this query becomes $\mathcal{S} = \{t_1, t_5, t_7\}$, and we want to cover this set by using minimum number of sets (which correspond to index servers) among the possible sets we can use for the selection of replicas: $\mathcal{S}_{IS_1} = \{t_1, t_7\}, \mathcal{S}_{IS_2} = \{t_1, t_7\}, \mathcal{S}_{IS_3} = \{t_5\}$, and $\mathcal{S}_{IS_4} = \{t_1, t_5, t_7\}$. Obviously, the optimum value is one, which is the selection of the set $\mathcal{S}_{IS_4}$ (thus the index server IS_4) to answer q_2.

Based on the mentioned observation, in the set-cover-based scheduling heuristic, we first schedule the non-replicated terms of a query. Then, replicated terms of this query are checked whether any of them has replicas in the index servers that are going to be used for the non-replicated terms. If so, these replicas are used. If there remain any replicated term(s) which cannot be scheduled using the method mentioned above, the scheduling of such replicated terms can be reduced to the set cover problem. Since this problem is NP-complete, a simple heuristic [55] is adopted to obtain a schedule for the remaining replicated terms. In each iteration, this heuristic simply selects the index server that contains the largest number of uncovered replicas so far, and then removes the currently covered replicas from all index servers. This process is repeated till there remains no uncovered replicated terms. This heuristic has an approximation ratio of $\ln(n)+1$ [55], where

n is the total number of elements in the ground set constructed for this query.

Fig. 4.5 illustrates the schedule obtained using set-cover-based scheduling on the replicated partition and the queries given in Fig. 4.4. Recall that the central broker maintains a map structure which is also shown in the figure. For $q_1 = \{t_1, t_2\}$, after scheduling non-replicated term t_2 to IS_1, since the replicated term t_1 can be scheduled to the index server(s) selected for the non-replicated term(s), t_1 is scheduled to IS_1, and the subqueries are formed respectively because there are no more terms to schedule. Since $q_2 = \{t_1, t_5, t_7\}$ terms which are all replicated, we use the above-mentioned heuristic for scheduling. As seen in the figure, IS_4 contains the largest number of replicas. Thus, in the first iteration, IS_4 is selected. After using replicas from IS_4, there remain no more replicas to cover, so we are done for q_2. Using the set-cover-based scheduling for the remaining two queries in a similar manner, we schedule q_3 to IS_2 and IS_4, and q_4 to IS_3 and IS_4.

4.5.2 Dynamic Load Balancing

In this scheme, the central broker makes the scheduling decision using dynamic information about the index servers. As in set-cover-based scheduling, we use the flexibility provided by the replicas of the replicated terms while scheduling. However, instead of scheduling to minimize the number of index servers involved in answering a query, the central broker schedules replicated query terms to the index servers with the *current* minimum load. To do this, it is enough for the central broker to maintain a simple array of size K that contains information about how many subqueries exist in the queue of each index server at a given time. Using this information, the central broker can dynamically schedule the replicated terms of a query to minimally loaded index servers. In this way, it may be possible to achieve a better load balance (on disk IOs, communication volume, etc.) than the set-cover-based scheduling, although it is likely that this scheme will incur a higher communication volume.

4.5.3 Hybrid Scheduling

Set-cover-based scheduling may suffer from high load imbalance whereas dynamic scheduling can suffer from high communication volume. An idea would be to use both scheduling heuristics by assigning certain weight of importance to them. This allows us to use a mixture of these heuristics by following a trade-off between the benefits and drawbacks of them. This hybrid algorithm may have a higher communication volume and a better load balance than the set-cover-based scheduling, and a worse load balance and a lower communication volume than the dynamic scheduling. However, by obtaining a trade-off between them, we expect the hybrid scheduling to have a lower average response time and higher throughput compared to both scheduling heuristics.

Both the set-cover-based and dynamic scheduling heuristics provide an ordered list of index servers as output. The order of an index server in these ordered lists is called its *rank*. In our approach, we use the ranks provided by both of these heuristics to schedule terms of a query to index servers. For a term of a query, let r_S and r_D be one-to-one and onto functions that return the rank of each index server for the set-cover-based and dynamic scheduling heuristics, respectively. We combine these two ranks using a hybrid scheduling parameter α as follows:

$$rank(i) = \alpha \times r_S(i) + (1 - \alpha) \times r_D(i), \text{ for } 1 \leq i \leq K,$$

where we select the index server with the smallest rank. After selecting an index server, necessary data structures are updated and this process is repeated till all terms of the given query are scheduled. If $\alpha = 1.0$, hybrid scheduling is equivalent to the set-cover-based scheduling, whereas if $\alpha = 0.0$, it is equivalent to dynamic scheduling. In order to identify good α values, we use parameter sweep techniques for both `AND` and `OR` document matching logics.

4.6 Experimental Results

In this section, we conduct experiments for comparing the proposed (replicated) partitioning schemes and scheduling heuristics. For this purpose, we implemented a real-time parallel query processing system called `repl-ABCServer` whose details are explained in Section 4.3. In our experiments, we first identify good hybrid scheduling parameters for both `AND` and `OR` logics, and then using these parameters, we conduct extensive experiments for varying number of index servers and replication amounts. The proposed schemes and heuristics are mainly tested for two important performance metrics, the average response time of a query (ART), and the number of queries processed per second (THR), the throughput. We also present results for other measures such as average number of processors (ANP) involved in answering a query, imbalance values and communication volumes.

4.6.1 Experimental Setup

The dataset used in the experiments is crawled from USA edu sites and consists of $1,883,037$ documents and $787,221,688$ terms with $3,325,075$ of them being distinct. From this dataset, $20,000$ synthetic queries are generated to be used as the query log for the (replicated) partitioning schemes we investigate. The generated queries follow a zipfian distribution and each query contains two to four terms since approximately 80% of all queries have four or less terms [56] and 65% of them have two to four terms. We exclude single term queries since scheduling of single term queries does not change the overall communication volume. These queries can easily be scheduled by taking the load balance concerns into consideration during query processing. The query log used for (replicated) index partitioning schemes is further utilized in the evaluation of these schemes using `repl-ABCServer`. In other words, after obtaining a (replicated) partition of the inverted index with the help of the query log using one of the methods given in Section 4.4, `repl-ABCServer` is run to answer the queries in the same query log with one of the scheduling heuristics given in Section 4.5. To measure

the peak performance of the system, we use $5,000$ arbitrary warm-up and cool-down queries (apart from $20,000$ queries) whose statistics are not included in the results.

The experiments are conducted on a homogeneous cluster environment that consists of 34 nodes connected with a 1 GB Ethernet. We use one of these nodes as the receptionist, one of them as the query submitter and the remaining nodes as the index servers. Each node has a single processor with 3.00 GHz of clock speed, 2 GB RAM, and 1 MB cache. The query submitter simulates 50 users that concurrently submit queries to the receptionist. It is assumed that a user does not submit another query till it receives the answer of the last query it submitted.

`repl-ABCServer` is implemented in C and uses MPI for passing messages in the cluster. It is capable of processing queries over a replicated and term-partitioned inverted index. It can process queries in `AND` and `OR` document matching logic and supports the proposed query scheduling heuristics given in Section 4.5 via parameters provided during initialization of the system.

In our experiments, we evaluate four basic (replicated) index partitioning schemes described in Section 4.4: `BP` (Bin-packing-based Index Partitioning), `BP+MF` (Most Frequent Term Replication), `PaToH` (Hypergraph-partitioning-based Index Distribution), and `rpPaToH` (Replicated Hypergraph-partitioning-based Index Distribution). These schemes are tested for four different number of index servers $K = 4, 8, 16$, and 32. For testing replicated partitioning schemes, three replication amounts are considered, no replication, 10% replication, and 25% replication. Note that the replication amount stands for the percentage of terms and their inverted lists that are replicated. When replication is involved, its amount is indicated in parenthesis with the used scheme, such as `rpPaToH` (10%), meaning `rpPaToH` is used for obtaining a replicated partition with 10% replication amount. The imbalance value is set to 20% both for `PaToH` and `rpPaToH`.

4.6.2 Selection of the Hybrid Scheduling Heuristic Parameter

The proposed hybrid scheduling heuristic which combines the set-cover-based and the dynamic scheduling heuristics is tested for varying α values. Recall that, when $\alpha = 1.0$, hybrid scheduling is equivalent to the set-cover-based scheduling and when $\alpha = 0.0$, it is equivalent to the dynamic scheduling. In our experiments, for both `AND` and `OR` document matching logics, α is varied from 0.0 to 1.0 with 0.1 increments to observe the trade-off between the minimization of communication volume (set-cover-based) and the minimization of load imbalance (dynamic), and these objectives' impacts on the evaluated metrics. Six schemes are evaluated for ART, THR, and ANP metrics: `BP`, `BP+MF` (10%), `BP+MF` (25%), `PaToH`, `rpPaToH` (10%), and `rpPaToH` (25%). Only the results for $K = 16$ are presented here since the results for other K values (4, 8, and 32) exhibit similar characteristics. Whenever performance of a scheme is mentioned, it is actually meant the ART and THR metrics, and not the ANP metric. In addition, since a lower ANP value does not necessarily indicate a better performance, only the ART and THR metrics are considered for the selection of the hybrid scheduling parameter. ANP values are presented to illustrate how good `PaToH` and `rpPaToH` schemes optimize their objectives and what kind of a relationship exists between the scheduling heuristics and the ANP values.

The results obtained by running `repl-ABCServer` with the above-mentioned configuration are illustrated in Fig. 4.6. As seen in the figure, as expected, the performances of `BP` and `PaToH` do not change as α varies since there is no replication in these schemes and this means that there is no flexibility during the scheduling process. `PaToH` performs a little better than `BP` which can be attributed to its better exploitation of the query log.

An analysis of the ART and THR results presented in Fig. 4.6 reveals that replication is indeed beneficial: `BP+MF` schemes always perform better than `BP` and `rpPaToH` schemes always perform better than `PaToH`. As the replication amount increases, replicated partitioning schemes obtain lower ART and higher THR values. If we compare `BP+MF` and `rpPaToH`, `rpPaToH` outperforms `BP+MF` for almost

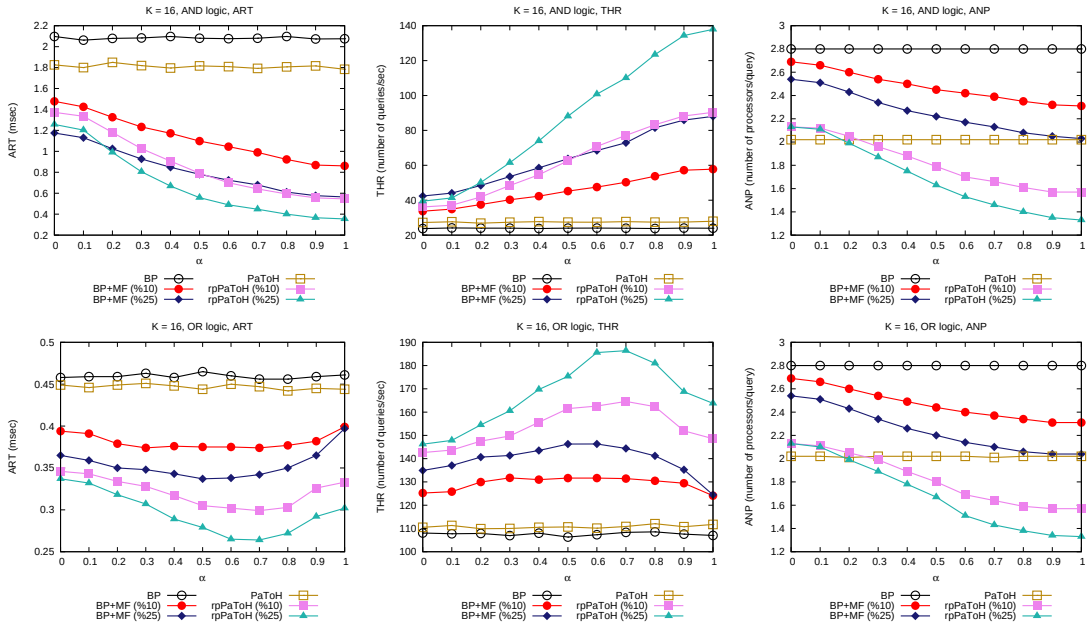


Figure 4.6: The average response time, throughput, and average number of processors for AND and OR document matching logics for 16 index servers ($K = 16$) and varying α.

all α values in both AND and OR logic. This is mainly due to two reasons: (i) rpPaToH tries to minimize the communication overhead while balancing the computational load whereas BP+MF only considers computational load balance, and (ii) rpPaToH is able to perform replication in a finer granularity compared to most frequent term replication.

As seen in Fig. 4.6, the OR logic obtains lower ART and higher THR values compared to AND logic for all schemes. This is mainly due to two reasons: (i) AND logic incurs higher communication volume compared to OR logic since we do not limit PAS sizes in AND logic while we do in OR logic (see Section 4.3.2 for a detailed explanation of this), and (ii) the receptionist in AND logic usually has to perform more computation since the PASs it receives in AND logic are usually longer. Thus, it can be said that the receptionist is more loaded in AND logic which throttles the performance of the system.

The first row of Fig. 4.6 presents the results for AND logic. For all replicated

partitioning schemes, the best results (ART and THR) are observed when $\alpha = 1.0$, which means the standalone set-cover-based scheduling performs the best for `AND` logic. This is basically due to high communication overhead incurred in `AND` logic which makes the receptionist a bottleneck. The set-cover-based scheduling aims to minimize this overhead by minimizing the number of index servers involved in answering a query and consequently reducing the load of the receptionist. Hence, it can be said that the set-cover-based scheduling is a better alternative for `AND` logic since it reduces the communication overhead so dramatically that the load balancing issues become secondary. Thus, in the rest of the experiments, α is set to 1.0 for `AND` logic.

The second row of Fig. 4.6 presents the results for `OR` logic. For almost all replicated partitioning schemes, the best results (ART and THR) are observed when α is around 0.6, which implies that a combination of set-cover-based and dynamic scheduling is optimal. This may be attributed to the following facts. Recall that, in `OR` logic, as opposed to `AND` logic, we take the union of the inverted lists instead of intersecting them and we limit the PAS sizes. Thus, the PASs do not incur as much communication volume as they do in `AND` logic. For this reason, it can be suggested that in `OR` logic, up to the point where α is around 0.6, the receptionist constitutes a bottleneck due to communication overhead whereas for higher α values, the index servers constitute a bottleneck due to load imbalance. For `OR` logic, setting $\alpha = 0.6$ for hybrid scheduling heuristic strikes a good balance between reducing communication overhead of the receptionist and balancing the index server loads. Thus, in the rest of the experiments, α is set to 0.6 for `OR` logic.

As seen in the third column of Fig. 4.6, ANP values of all replicated partitioning schemes steadily decrease as α gets closer to 1.0. This is because the set-cover-based scheduling heuristic's sole purpose is to reduce the number of index servers involved in answering queries using the flexibility provided by the replication. Since this flexibility is not present in `BP` and `PaToH`, the ANP values do not change as α varies for these two schemes. It is evident that the higher the replication amount, the lower the ANP values in replicated partitioning schemes. This can be seen by comparing the `BP`-based and `PaToH`-based schemes among

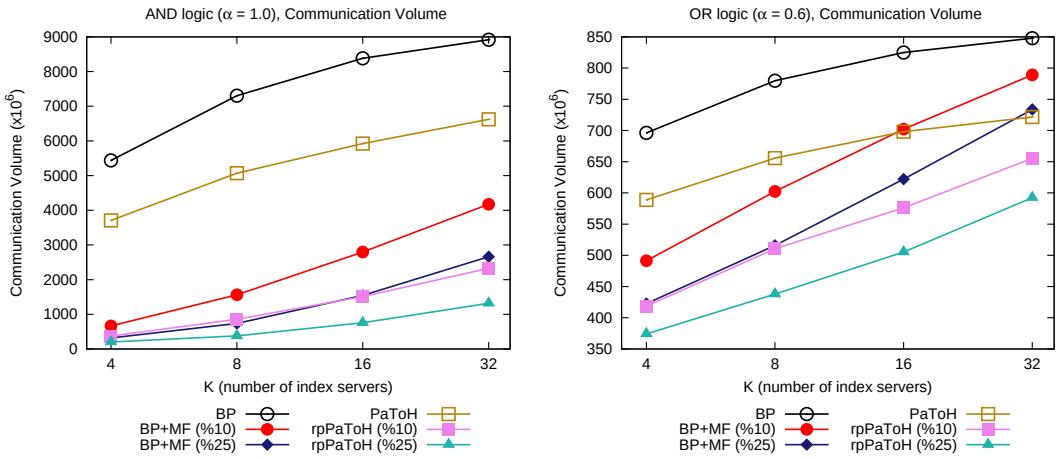


Figure 4.7: Communication volume values for `AND` and `OR` document matching logics and varying number of index servers (K).

themselves. This is expected since when the replication amount increases, the opportunity for reducing the number of processors involved in answering a query also increases. Note that the ANP figures for `AND` and `OR` logic are identical since the scheduling process does not depend on the used document matching logic.

`PaToH`'s ANP values in the third column of Fig. 4.6 are quite noteworthy since its objective is the minimization of the ANP. Although `PaToH` does not utilize any replication, it can outperform replicated partitioning scheme `BP+MF` in obtaining lower ANP values. Note that `PaToH` can also outperform `rpPaToH` when dynamic scheduling is used ($0.0 \leq \alpha \leq 0.2$) since this scheduling heuristic may opt to increase ANP in favor of reducing imbalance. `PaToH` obtains lower ANP values due to its superior partitioning quality compared to both `rpPaToH` schemes at low α values. However, lower ANP values do not indicate better ART and THR values as seen in Fig. 4.6 where `PaToH` is outperformed by most of the other schemes. This is due to the high communication imbalance of `PaToH` (Section 4.6.5).

4.6.3 Communication Volume

This section presents the communication volume values of six schemes BP, BP+MF (10%), BP+MF (25%), PaToH, rpPaToH (10%), and rpPaToH (25%) for both AND ($\alpha = 1.0$) and OR ($\alpha = 0.6$) logic with varying number of index servers, $K = 4,8,16$, and 32. The basic unit of communication volume for the results given in Fig. 4.7 is an accumulator entry ($a_j = (d_j, s_j)$) that consists of a four byte document id field and a four byte score field, making up to eight bytes. The communication volume incurred by the receptionist while distributing subqueries to index servers is not included since this quantity is the same for all tested schemes.

The communication volume values are illustrated in Fig. 4.7. The first obvious observation is that all schemes incur lower communication volumes in OR logic compared to AND logic, which is basically due to limitation of PAS sizes in OR logic. Another immediate observation is that as K increases, the communication volume incurred by any scheme also increases. This is due to the increase in the average number processors (ANP) (see third column of Fig. 4.6), which implies a query is being processed at more index servers as K increases. Since the number of index servers involved in answering a query increases, the communication volume also increases regardless of the used document matching logic.

As seen from Fig. 4.7, rpPaToH (25%) achieves the lowest communication volume for all K in both AND and OR logics. For the replicated partitioning schemes, the communication volume of rpPaToH (10%) and rpPaToH (25%) are lower than those of BP+MF (10%) and BP+MF (25%), respectively. This shows that rpPaToH, when coupled with the right hybrid scheduling heuristic, is better at utilizing replication to reduce the communication volume compared to BP+MF. PaToH outperforms BP+MF (10%) at $K = 16$, and both BP+MF (10%) and BP+MF (25%) at $K = 32$ in OR logic, since the partitions obtained by PaToH incur lower communication overhead compared to BP+MF that utilizes replication and makes a certain majority of its scheduling decisions in order to reduce the communication volume. This is not observed in AND logic since the communication volume reduction obtained by the replication using scheduling heuristics are so significant that PaToH cannot even get close to the replicated schemes.

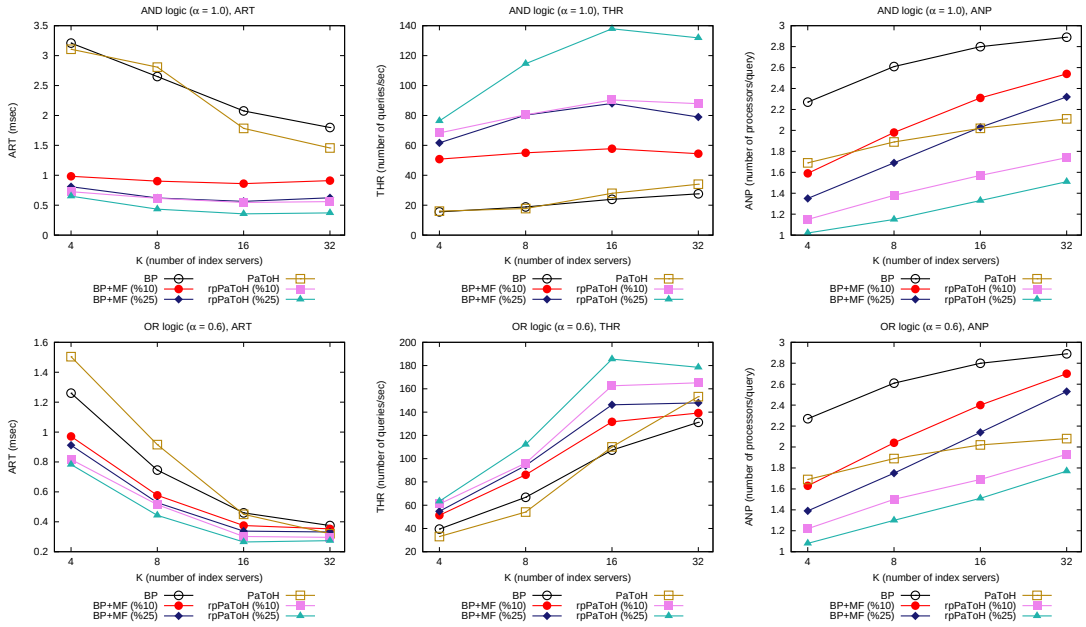


Figure 4.8: The average response time, throughput, and average number of processors for `AND` ($\alpha = 1.0$) and `OR` ($\alpha = 0.6$) document matching logics for varying K values.

4.6.4 Weak Scaling Performance

This section presents the evaluation of six (replicated) partitioning schemes, `BP`, `BP+MF` (10%), `BP+MF` (25%), `PaToH`, `rpPaToH` (10%), and `rpPaToH` (25%) in terms of ART, THR, and ANP metrics for varying number of index servers, $K = 4, 8, 16, 32$. These schemes are tested for `AND` (with $\alpha = 1.0$) and `OR` (with $\alpha = 0.6$) document matching logics. We present the weak scaling results, i.e., the data size and the number of processes that concurrently submit queries are kept constant while K is varied.

The results of the experiments for the above-mentioned configuration are illustrated in Fig. 4.8. All schemes are expected to perform better (ART and THR) since the amount of work per index server decreases as K increases. This expected pattern is observed up to $K = 16$. However, when K increases from 16 to 32, the ART and THR values of the replicated schemes stabilize. Our detailed analysis reveals that when K increases from 16 to 32 in replicated schemes, the

computation, communication and IO times of index servers roughly halve as expected. However, the idle times of the index servers increase significantly, forming up to 60% of the servers' total running time. This indicates that the (replicated) partitioning schemes successfully distribute the index server loads, however, the receptionist cannot cope with the increase in the concurrently received PASs. Thus the receptionist becomes a bottleneck after $K = 16$ and determines the performance of the system for the replicated partitioning schemes. It should be noted that there is also a significant increase in idle times of the index servers for `BP` and `PaToH` as well. However, since the amount of communication volume, and thus the index server loads for these schemes are quite high, increasing the number of index servers still improves the performance of the system by reducing the index server loads.

As seen from ART and THR metrics in Fig. 4.8 up to $K = 16$, the performances of all schemes increase more dramatically in `OR` logic compared to `AND` logic. This is because when K is doubled, the communication volumes in `AND` logic incurred by all schemes increase by a higher factor than they do in `OR` logic (see Fig. 4.7). Therefore, the schemes in `OR` logic obtain a better performance speedup. For example, when K varies from 4 to 8, the communication volume of `rpPaToH` (10%) increases by a factor of 2.31 in `AND` logic whereas it increases only by a factor of 1.22 in `OR` logic, and when K varies from 8 to 16, the communication volume of `rpPaToH` (10%) increases by a factor of 1.76 in `AND` logic whereas it increases only by a factor of 1.13 in `OR` logic. Thus, `rpPaToH` (10%) obtains a better speedup in `OR` logic as seen in ART and THR metrics of Fig. 4.8.

The third column in Fig. 4.8 displays the ANP results for varying K values. For all schemes, as K increases, the ANP values of all schemes increase steadily. That is because the data size is kept constant while K increases, and as K increases, it is more probable that the terms requested by a query will not belong to the same index server. Note that the ANP figures for `AND` and `OR` logic are not identical as for the case in Fig. 4.6 since the hybrid scheduling parameters used for these document matching logics are not the same.

Table 4.2: Communication volume, disk access, and disk IO imbalance (%) values for varying K and `AND` and `OR` document matching logics.

		Imbalance (%)											
		Communication Volume				Disk Access				Disk IO			
Scheme	Logic	$K=4$	$K=8$	$K=16$	$K=32$	$K=4$	$K=8$	$K=16$	$K=32$	$K=4$	$K=8$	$K=16$	$K=32$
BP	AND	5.4	13.7	36.9	82.5	0.6	1.5	3.1	4.3	4.6	15.5	37.8	82.1
	OR	0.4	1.3	4.5	7.6	0.6	1.5	3.1	4.3	4.6	15.5	37.8	82.1
BP+MF (10%)	AND	1.3	18.8	36.9	82.5	8.9	13.5	17.2	9.1	18.0	32.6	59.9	46.6
	OR	0.2	0.2	0.9	6.9	1.4	3.0	1.9	3.6	2.6	6.9	14.3	31.3
BP+MF (25%)	AND	0.5	5.6	29.5	35.2	4.0	18.4	22.2	20.0	5.6	37.8	61.7	46.6
	OR	0.4	0.2	1.4	1.3	0.8	1.4	3.1	3.7	3.3	3.4	10.4	23.6
PaToH	AND	106.6	159.9	122.3	170.8	20.1	19.0	8.5	4.8	100.2	138.6	111.8	141.6
	OR	26.4	33.5	24.5	23.5	20.1	19.0	8.5	4.8	100.2	138.6	111.8	141.6
rpPaToH (10%)	AND	1.2	1.9	1.8	1.6	6.4	36.4	14.6	27.0	8.6	8.7	7.2	12.5
	OR	0.3	23.0	3.0	4.4	1.7	23.8	7.6	17.8	2.0	13.7	26.1	55.3
rpPaToH (25%)	AND	0.5	1.2	1.6	4.0	8.3	34.0	11.4	23.5	7.0	21.5	20.3	18.8
	OR	0.2	12.0	0.3	0.4	4.3	14.0	7.5	8.6	6.5	5.0	11.4	22.1

4.6.5 Imbalance

In this section, communication volume, disk access, and disk IO imbalance values of six (replicated) partitioning schemes, BP, BP+MF (10%), BP+MF (25%), PaToH, rpPaToH (10%), and rpPaToH (25%) are presented. The results for AND and OR document matching logics are reported for four K values, 4, 8, 16, and 32. As mentioned, PaToH's and rpPaToH's imbalance in partitioning are both set to 20%. Note that both BP- and PaToH-based schemes balance on the term frequencies, where a term's frequency is given by the number of queries that contain this term (Section 4.4). Hence, this is equivalent to balancing the *number of disk accesses* (not the disk IO) performed by each index server. There is a rough relation between the number of disk accesses and the disk IO performed by an index server. The strength of this relation decreases with increasing variance in the inverted list lengths.

Table 4.2 displays imbalance values for communication volume, disk access and disk IO. As seen from table, all schemes have generally lower imbalance in OR logic compared to AND logic. This is because, for communication volume, the PAS sizes are limited to a fixed size in OR logic, which makes the sizes of the sent messages by the index servers roughly equal. OR logic generally obtains lower imbalance in disk access and disk IO since the scheduling heuristic parameter α is set to 0.6 for OR logic whereas it is set to 1.0 for AND logic. This implies that the OR logic uses a certain amount of dynamic information in its scheduling decisions whose aim is to balance the load of the index servers.

Another immediate observation that can be inferred from Table 4.2 is that BP-based schemes generally obtain better disk access imbalance compared to their PaToH-based counterparts since obtaining a balance on disk access is the *objective* of BP-based schemes whereas it is the *constraint* of PaToH-based schemes. Note that for some experiment instances, rpPaToH's disk access imbalance is greater than 20% although it is set to obtain replicated partitions up to this value. This is due to the different utilization of replication in computing imbalance during partitioning and making scheduling decisions.

In AND logic, especially in partitioning schemes that do not utilize replication (BP and PaToH), we can claim that the imbalance in disk IO would probably be reflected in communication volume imbalance. This is due to the fact that since we have short queries consisting of two to four terms, it is very probable that if all terms of a query are not processed at the same index server, each index server will process a single term of this query. Note that in AND logic, processing of a single term incurs the same IO and communication costs. High disk access imbalance of PaToH causes high disk IO imbalance, which in turn causes high communication imbalance. This relation between disk access, disk IO, and communication imbalance is not as prominent when there is replication since replication can be utilized for balancing loads of the index servers. For BP and PaToH, disk access and disk IO imbalance for both AND and OR logics are the same since although α is set to different values for both logics, there is no freedom in scheduling a query term to different index servers. Thus, the query terms are assigned to the same index servers in both AND and OR logic and the same disk access and disk IO imbalance values are obtained.

4.7 Related Work

In this section, we discuss the most commonly used approaches for replication in parallel information systems. In [57], the authors analyze caching and partial replication to improve the performance of parallel information retrieval systems. In their approach, they replicate the documents requested by previous queries (thus, using a query log) to a distinct server, which is called the partial replica. This partial replica is further used to answer the future queries if possible. In their work, they show that the replication is able to reveal access locality of the queries.

In [58], the performance of clustered and replicated IR systems are investigated. In their work, a document-based index distribution is assumed and a simulation environment is used to compare these two systems. In their approach, a replicated system consists of identical distributed systems, where each distributed

system contains all collection and the brokers decide which replica will be used for a given query. On the other hand, a clustered system consists of disjoint sets of documents, where each cluster can be distributed or replicated. Their experiments show that the clustered system does not outperform the replicated system and the brokers and the network can be bottleneck in both of these systems.

In Google cluster [59], which consists of hundreds of commodity machines, replication is mainly used for performance (i.e. throughput) and availability. The index is partitioned in a doc-based manner and replication is used in all levels of the cluster architecture including the hardware-based solutions (RAID), the cluster itself and across the clusters.

The authors of [52] use the query log to obtain information about term frequencies which is further utilized for the distribution and replication of terms to index servers. Their term distribution approach is based on a heuristic that is generally used for the bin-packing problem. In their replication method, they replicate a certain amount of most frequent terms and their inverted lists to all index servers. Performing term distribution and replication as mentioned, they show that they can achieve better load balancing for term-based partitioned indexes.

Another recent research utilizing query logs is [53]. As in [52], they first distribute the terms based on a bin-packing heuristic using index servers as bins and terms as items. For the remaining terms which do not appear in query log, they distribute them to the index servers so that each index server possesses approximately equal number of terms. They also investigate replication by replicating a certain percentage of most frequent terms to all index servers. Thus, by using query logs to obtain a better term distribution and replication, they are able to achieve improvements in response time and throughput.

Chapter 5

Partitioning and Replication for Social Networks

In this chapter, we propose to model the interactions among social network users via a novel temporal activity hypergraph and then perform a replicated partitioning of this temporal hypergraph to deduce a user partitioning. Our aim is to minimize the communication overheads of parallel systems used in social networks. We focus on the Twitter application, but we believe the proposed modeling can be applied in defining the multi-way interactions of other social networking systems as well.

5.1 Introduction

Social networks have fast-growing, ever-changing dynamic structures and strict availability requirements which led to development of non-orthodox solutions such as NoSQL systems. These systems use data partitioning and replication to achieve scalability and availability and this is generally achieved via hash-based partitioning and random replication of data. This approach ensures availability and a certain extent of scalability, however, since it ignores the relations among the data, it may lead to unnecessary replications and communication loads while

responding to user queries.

In order to alleviate these deficiencies, partitioning of friendship graphs [60, 61] and replication of the data associated with the users that are neighbor to partition borders have been proposed [61, 62]. Unfortunately, since friendship graphs exhibit power-law distributions and tightly-coupled community structures, partitioning of these graphs are quite difficult and the replications proposed by these models lead to excessive replication amounts. Furthermore, these graphs do not consider the temporal locality existing in user actions. In retrospect, a partitioning scheme based on partitioning of activity graphs involving simple time relations have been proposed [63], but replication in such graphs has not been investigated. Furthermore these graph-based schemes try to capture the interactions between social network users via two-way relations. However, the most common queries in social networks such as requesting the latest Tweets of users being followed or requesting the latest news-feeds of Facebook friends involve fold-like operations that require gathering of data from multiple users to a single user.

In this chapter, we propose a thorough remodeling of the partitioning and replication schemes for social networks. To this end, we focus on the Twitter application and we propose to model the multi-way relations between user actions in Twitter via a novel temporal activity hypergraph. Then, we propose to perform a replicated partitioning of this hypergraph (via utilizing a state-of-the-art replicated hypergraph partitioning tool that performs partitioning and replication together, thus combining the benefits of these two schemes) and replicate and distribute the user data according to the result of this replicated partitioning. We compare the communication requirements of our replicated partitioning scheme with a distributed hashing based partitioning replication scheme. Our results indicate that proposed temporal hypergraph model can significantly increase the number of queries that can be locally answered, thus reducing the communication overhead.

5.2 Background

5.2.1 Hypergraph Partitioning

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as a set $\mathcal{V}$ of vertices and a set of $\mathcal{N}$ nets (hyperedges), where each net connects a number of distinct vertices. The vertices connected by a net n_j are said to be its pins and denoted as $\text{Pins}(n_j)$. A cost $c(n_j)$ is assigned as the cost of a net $n_j \in \mathcal{N}$. In both graphs and hypergraphs, multiple weights $w^1(v_i), w^2(v_i), \ldots, w^M(v_i)$ may be associated with a vertex $v_i \in \mathcal{V}$.

$\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ is said to be a K-way partition of a given hypergraph $\mathcal{H}$ if each vertex part $\mathcal{V}_k$ of Π is a nonempty subset of $\mathcal{V}$, parts are pairwise disjoint, and the union of the K parts is equal to $\mathcal{V}$. The K-way HP problem can be defined as finding a K-way vertex partition Π that optimizes a partitioning objective defined over the edges and nets that connect more than one part while satisfying a given partitioning constraint.

In HP problem, the partitioning constraint is to satisfy multiple balance criteria on part weights, i.e.,

$$W^m(\mathcal{V}_k) \leq W^m_{\text{avg}}(1 + \epsilon^m), \text{for } k = 1, \ldots, K;\ m = 1, \ldots, M. \tag{5.1}$$

Here, for the mth constraint, the weight $W^m(\mathcal{V}_k)$ of a part $\mathcal{V}_k$ is defined as the sum of the weights $w^m(v_i)$ of the vertices in $\mathcal{V}_k$, W^m_{avg} is the average part weight, and ϵ^m is the maximum allowed imbalance ratio.

In a partition Π of hypergraph $\mathcal{H}$, a net is said to connect a part if it has at least one pin in that part. The connectivity set $\Lambda(n_j)$ of a net n_j is the set of parts connected by n_j. The connectivity $\lambda(n_j) = |\Lambda(n_j)|$ of a net n_j is the number of parts connected by n_j. A net n_j is said to be cut if it connects more than one part (i.e., $\lambda(n_j) > 1$) and uncut otherwise. In the HP problem, the partitioning objective is to minimize the connectivity-1 metric

$$\chi(\Pi) = \sum_{n_j \in \mathcal{N}_{\text{cut}}} c(n_j)(\lambda(n_j) - 1), \tag{5.2}$$

defined over the set $\mathcal{N}_{\text{cut}}$ of cut nets.

HP problem is known to be NP-hard [64, 65]. However, there are successful HP tools (e.g., hMETIS [66], and PaToH [67]) that implement efficient and effective heuristics.

5.2.2 Replicated Hypergraph Partitioning and rpPaToH

In the *Replicated Hypergraph Partitioning Problem*, we are given an undirected hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ an imbalance ratio ϵ, and a replication ratio ρ and we want to find a K-way covering subset of $\mathcal{V}$, $\Pi^R = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ that minimizes the cutsize (Eq. 5.2) while satisfying the following constraints.

- Balancing constraint: $W_{max} \leq (1 + \epsilon)W_{avg}$, where $W_{max} = max_{1 \leq k \leq K} W(\mathcal{V}_k)$ and $W_{avg} = (1+\rho)W(\mathcal{V})/K$.
- Replication constraint: $\sum_{k=1}^{K} W(\mathcal{V}_k) \leq (1 + \rho)W(\mathcal{V})$

rpPaToH [68, 5] is a novel tool that performs replicated hypergraph partitioning. rpPaToH is capable of replicating vertices of an undirected hypergraph during partitioning in order to improve a target objective under given balancing and replication constraints.

5.2.3 Twissandra: An Educational Twitter Clone

Twissandra [69], is an example project that provides a fully-working Twitter clone based on Cassandra. The Twissandra schema used to represent Twitter data model is simple but in terms of partitioning and distribution decisions scaling Twissandra carries most fundamental problems observed in scaling Twitter.

The Twissandra data model consists of six column families:

- `USER`: Holds user information; the key for each row is the username and columns hold user details such as passwords.
- `FRIENDS`: Holds the users that are followed by a user (friends); the key for each row is the username and columns are the usernames of the friends, which are the users followed by the user in the row key.
- `FOLLOWERS`: Holds the followers of a user; the key for each row is the username and columns are the usernames of the users which follow the user in the row key.
- `TWEET`: Holds the tweets; the key for each row is a unique tweet ID and columns are the tweet body and the username of the tweeting user.
- `TIMELINE`: Holds the tweets of friends a user is following; the key for each row is the username and column names are timestamps and column values are tweet IDs.
- `USERLINE`: Holds all the tweets by a given user; the key for each row is the username and column names are timestamps and column values are tweet IDs.

,

Using this data model, it is possible to model many functionalities in Twissandra. However, the main operations we will investigate here will be the creation of a tweet (a tweet write) and a getting the timeline of a specific users tweets (viewing/reading tweets in homepage). The first operation is performed when a user tweets and it requires insertion of a tweet to the `TWEET` cf, the addition of the unique tweet ID into the `USERLINE` cf of the tweeting user, and the addition of the unique tweet ID into the `TIMELINE` column families of the followers of the tweeting user. The second operation is performed when a user connects to the Twitter and checks the latest tweets of his friends. Here, a "friend" is somebody that a user follows. It requires a lookup for the latest tweet IDs of a user in his/her respective row at the `TIMELINE` cf and then retrieval of the tweets for those tweet IDs from the `TWEET` cf. We choose to model these two operations since they are

performed quite frequently in Twitter and they are costly operations. Since we do not have user access logs in our datasets and only user tweet information, we make the assumption that each user reads his/her recent tweets before tweeting.

5.3 Temporal Activity Hypergraph Model

In this section we describe how to construct the temporal activity hypergraph model $\mathcal{H}_{\text{act}}^{\text{tmp}} = (\mathcal{V}, \mathcal{N})$ from a log of a social graph among a set of social network users and a log of interactions (tweets in this case) among these users. A replicated partitioning of this hypergraph will be used in the replicated placement of these data in a parallel system serving the Twissandra application and this placement will be inspected in terms of its' efficiency in serving new tweet read requests of the sort where a user reads his/her friends latest tweets.

We assume that the logs contain information on the timing of actions. Just like in [63], we divide our logs into time periods and utilizing the logs of previous periods, try to figure the pattern and frequency of actions that will happen in the current time period and partition/replicate data according to this prediction. The time periods can be months, weeks, days, or even hours. We value actions that occurred in recent periods more than older actions and value all actions that occurred in the same period equally. The selection of these time periods determine the frequency of partitioning.

Our replicated partitioning scheme proposes a horizontal partitioning of the Twissandra column families. In particular, we propose a user partitioning scheme and this user partitioning induces a partitioning on all cfs of Twissandra. It is easy to see that a partitioning on users easily induces a row-based partitioning on the `USER`, `FRIENDS`, `FOLLOWERS`, `TIMELINE`, and `USERLINE` column families since the row keys for all these cfs are the username. The partitioning of `TWEET` cf is performed according to the username of the tweeting user. In the end, each user's personal information, friends, followers, userline, timeline and tweets are stored on the same server(s).

5.3.1 Model Construction

In this section we describe the construction of our temporal activity hypergraph model $\mathcal{H}_{\text{act}}^{\text{tmp}} = (\mathcal{V}, \mathcal{N})$. In $\mathcal{H}_{\text{act}}^{\text{tmp}}$, for each user u_i there is a vertex $v_i \in \mathcal{V}$. A user request for the latest tweets of his/her friends is called an activity. For each activity a_j, where a user u_i requests for the latest tweets of his/her friends, there is a net $n_j \in \mathcal{N}$, which connects v_i and the vertices for the friends of user u_i. i.e., for an activity a_j of a user u_i,

$$\text{Pins}(n_j) = \{v_k : v_k = v_i \vee u_k \in friends(u_i)\}, \tag{5.3}$$

where $friends(u_i)$ indicate the set of users followed by user u_i. Note that two nets connecting the same set of vertices are called identical nets and such nets can be represented with a single net by adding the costs of the two nets. Note also that if the set $friends(u_i)$ of a user u_i does not change, all the activities of u_i will generate identical nets.

The weight $w(v_i)$ of a vertex v_i is set to reflect the amount of activity user u_i performs. That is, it is related to the number of times the user logs in to the system to check for tweets. The cost $c(n_j)$ of a net n_j is set to reflect the closeness of the activity to current time. The closer an activity to current time, the higher its cost. Temporality comes into play in setting vertex weights and net costs. Just like in [63], we use a decay function to set an order of precedence among activities in different time periods such that activities in recent periods have higher importance and thus, the costs of nets representing these recent activities have higher values and the weight of vertices representing users who are active in the recent periods are high.

5.3.2 Replicated Partitioning of the Model

Maintaining the partitioning constraint of balanced part weights is expected to balance the number of read requests received by the processors in the current time period, whereas the partitioning objective of reducing the cutsize minimizes the number of servers involved in answering a read query.

As an example to illustrate the construction of our temporal activity hypergraph model, consider the following sample log of activities that occurred in different time periods in the past. For the ease of demonstration, let us assume that our system displays only the most recent three tweets of their friends to users, all users are equally active and hence vertices have equal weights, and the decay factor is two (as in exponential smoothing), meaning that the net for an activity in a time period has half the cost of a net for an activity in the next period.

- activity a_1: in time period t_1 user u_1 connects and reads the tweets of users u_2, u_3 and u_4.
- activity a_2: in time period t_1 user u_2 connects and reads the tweets of users u_3, u_4 and u_5.
- activity a_3: in time period t_3 user u_2 connects again and this time reads the tweets of users u_3 and u_5 (it may be that friends of t_3 are not tweeting very frequently, so he may receive less than three tweets).
- activity a_4: in time period t_4 user u_4 connects and reads the tweets of users u_1, u_5 and u_6.
- activity a_5: in time period t_4 user u_9 connects and reads the tweets of users u_6, u_7 and u_8.

Fig. 5.1 shows the temporal activity hypergraph model for the above-given sample log of activities and Fig. 5.2 shows a 3-way partition of this temporal activity hypergraph model. In the figures, user vertices are represented via empty circles and activity nets are represented with dots. In Fig. 5.2, replicated vertices are indicated via dot-lined red circles. As equal user activities and hence unit vertex weights are assumed, in Fig. 5.2, the part weights for the three parts are equal to four. Since only net n_4 remains in the cut, the cut according to Eq. 5.2 would be $c(n_4)(\lambda(n_4) - 1) = 8 \times (3 - 1) = 16$.

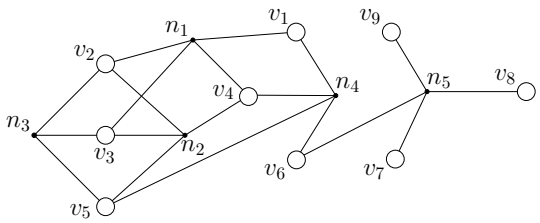


Figure 5.1: Temporal activity hypergraph model $\mathcal{H}^{\text{tmp}}_{\text{act}}$ for the sample log. Considering a decay factor of two, the costs of nets can be set as follows: $c(n_1) = c(n_2) = 1$, $c(n_3) = 4$, and $c(n_4) = c(n_5) = 8$.

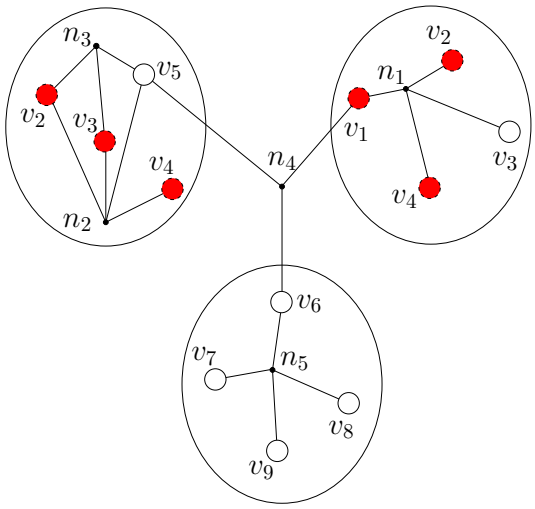


Figure 5.2: A 3-way replicated partition of the $\mathcal{H}^{\text{tmp}}_{\text{act}}$ for the sample log. The dashed, red-filled circles indicate replicated vertices.

5.4 Experimental results

In our experiments we made use of the Twitter dataset from [70]. This dataset was crawled from Twitter between October 2006 - November 2009 and there are a total of 465.107 distinct users in the dataset. Among these users there are 836.541 social relationships (follower, followed by, etc) and the dataset contains a total of 25.378.846 tweets. Within this dataset, we made use of 8.105.164 tweets that were made between October 2008 - September 2009. These tweets were made and viewed by 107.562 users.

For each tweet we generated a read query assuming that the tweeter views

Table 5.1: Comparison of cut values for DHT and RHP.

K	RHP (100% repl.)	RHP (200% repl.)	DHT (2-copy)	DHT (3-copy)
4	173.704	4.434	2.894.605	295.852
8	725.298	351.736	5.659.987	3.891.994
16	1.069.189	754.042	6.929.248	6.105.297
32	1.265.268	1.066.924	7.509.313	7.125.869
64	1.426.434	1.277.756	7.810.679	7.625.661
128	1.536.381	1.417.860	7.956.350	7.869.601
256	1.664.307	1.587.719	8.031.091	7.990.493
512	1.795.868	1.705.728	8.065.724	8.044.932
1024	2.120.683	1.954.846	8.085.462	8.071.713

his/her latest tweets prior to tweeting. The read queries are generated considering the interactions between Twissandra cfs as described in Section 5.2.3.

We constructed the temporal hypergraph model for the generated read queries as described in Section 5.3 and performed a replicated partitioning of this hypergraph as described in Section 5.3.2 We tried to assess the reduction in communication via our replicated hypergraph partitioning based replicated partitioning scheme (RHP) as compared to the circular-replication/distributed-hash-table (DHT) scheme generally applied in NoSQL systems. As communication metric, we counted the number of queries that can not be answered locally and requires coordination among the processors. We tested our system with 100% and 200% replication ratios and the DHT scheme with 2-copy and 3-copy replications.

As seen in Table 5.1, the RHP scheme significantly reduces the number of queries that requires coordination among processors. In other words, RHP increases the locality in queries in comparison to DHT. Another interesting observation we make from Table 5.1 is that, the positive effect of increasing replication ratio (e.g., from 100% to 200% r from 2-copy to 3-copy) diminishes with increasing number of processors. This is probably due to the fact that, as the number of processors increase, the data becomes very dispersed and it becomes difficult to gather data such that query responses can be generated locally. As seen in the table, for 1024 processors, even in the RHP scheme, around 25% of the queries require coordination.

5.5 Related Work

Large-scale data and fast data growth trends of many Web 2.0 applications supported the development of NoSQL (Not Only SQL) solutions as an alternative to RDBMS (Relational Database Management Systems) [71]. RDBMS are not designed to scale on multiple nodes, or rather they are not designed to scale cheaply, whereas NoSQL approaches are designed for i) cheap scalability, ii) fast read and write, iii) batch data processing and iv) easy column addition/removal [72]. On the other hand, they lack important helpful features such as SQL support and transaction reporting.

NoSQL databases can be grouped according to their data model, query properties, concurrency control, partitioning, replication and consistency features. According to data model, they can be grouped into four classes:

- Key-Value databases: In this data model, data is stored in maps or dictionaries, which are accessed via unique keys. Only way to access data is via these keys, which complicates value-based queries. Amazon Dynamo [10], Project Voldemort [73], Redis [74], Scalaris [75] and Membase [76] utilize this data model.

- Document Stores: In this data model, key-value pairs are stored in documents and each document has a unique key. Unlike key-value databases, this model supports value-based multi-attribute queries. CouchDB [77], MongoDB [78], Terra Store [79], and Riak [80] utilize this data model.

- Distributed Column Stores, C-Store: This data-model is optimized for accessing multiple rows for a given column. Note that this is orthogonal to classic RDBMS. Google BigTable [81], HBase [82] and HyperTable[83] utilize this data model. Cassandra [84] also uses C-Store data model with the addition of super-columns, an extra layer over columns that groups multiple columns that will be stored together.

- Graph Databases: This data model is designed for applications that require graph-traversal-type queries over highly-connected data. Neo4j [85],

GraphDB[86], Sesame [87], and BigData [88] utilize this data model.

There are recent studies indicating the deficiencies of the partitioning and replication methodologies used in NoSQL systems. [89] proposes a graph-partitioning-based database partitioning scheme for OLTP-type Web applications. In this method, data items are represented via nodes, transactions are modeled via edges, and the partitioning objective is minimizing the number of servers involved in processing a transaction. Unfortunately, the replication scheme proposed in [89] is quite costly since it requires K times replication of each data vertex prior to partitioning, K being the desired number of partitions.

[60] proposes alternative partitioning schemes based on graph-partitioning, modular-optimization and random partitioning. Partition qualities are justified in metrics such as the number of internal messages or dialogs and tests are performed over datasets collected from Twitter and Orkut. For small partition counts, graph-based approaches are shown to perform superior, whereas for large partition counts, modular optimization algorithms are shown to perform better. Interestingly, when the partition counts increase, random partitioning perform as good as graph-partitioning. [61] extends the works in [60] so that replication is also considered. Proposed replication scheme (one-hop replication) replicates all data items that are in partition boundaries. That is, data items that might be required in multiple servers are replicated to all of those servers. Unfortunately, this replication scheme enforces too much replication to be of practical use. [62] is an extension of these two studies with an alternative partitioning scheme. However, still the one-hop replication scheme is used for replication.

[63] propose graph-partitioning based models for efficient query processing in time-dependent social network queries such as collecting status updates of friends. Their activity prediction graph model enables handling of power-law relations that are consistently observed in almost all social network data features via producing lighter tailed interactions. Unfortunately, this study does not address the replication problem.

Chapter 6

Conclusions

In this thesis we proposed query-log-aware FM-like iterative improvement solutions to replicated declustering and replicated clustering problems. Chapters 2 and 3 are related with improving the performance of replicated declustering schemes, whereas Chapters 4 and 5 are related with improving the performance of parallel keyword-based search and Twitter applications via query-log-aware replicated clustering.

In Chapter 2, we proposed an effective K-way replicated declustering scheme that utilizes a given query distribution. To this end, we first proposed an iterative-improvement based two-way replicated declustering scheme, which iteratively improves the quality of a two-way replicated declustering. We recursively applied this two-way scheme to obtain a K-way replicated declustering. We then proposed an efficient and effective multi-way refinement scheme that can perform multi-way move and replication of data items. With this scheme, we further improved the quality of the obtained K-way declustering and improved the balance if possible. Presented results indicated the merits of utilizing query logs in partial and selective replication. We showed that the proposed scheme achieves much better results compared to state-of-the-art replicated declustering schemes, many times achieving optimal overall response time with less than 100% replication ratio.

In Chapter 3, we proposed an effective three-phase replicated re-declustering framework that utilizes a given query distribution and an exiting data item to server mapping to minimize query processing and data migration costs. The first phase of this framework generated a replicated declustering solution via utilizing the query logs. In this phase, any log-utilizing replicated declustering algorithm can be utilized. We chose our recursive replicated declustering algorithm described in Section 2.4.1 since it was shown to perform good. For the second phase, we proposed a weighted bipartite matching model that matches any replicated declustering solution to the servers in the system. This model finds the matching that causes the minimum number of data migrations. For the third phase, we proposed a multi-way refinement scheme that not only improves the query processing costs of the replicated declustering solution obtained in the first two phases, but also reduces the migration costs of the final mapping. To enable optimization of the two conflicting minimization objectives (query processing and migration cost minimization), we proposed an abstraction scheme that enabled us to represent data migrations as queries. This scheme also contains a mechanism for maintaining a balance between query processing cost reduction and migration cost reduction objectives. Using this scheme also enabled the usage of multi-way refinement scheme proposed in Section 2.4.2 for the solution of the replicated re-declustering problem. Obtained results indicated the merits of the proposed re-declustering scheme. Compared to the replicated declustering scheme we proposed in Chapter 2, the proposed framework generated results with significantly lower migration costs with slightly higher query processing costs.

In Chapter 4, we adopted our recently proposed replicated-hypergraph-partitioning-based approach for generating replicated, term-partitioned indexes [5] and evaluated the performance of this approach against state-of-the-art partitioning and replication schemes. We also discussed various scheduling schemes that are required when replication is involved. We investigated these schemes on a realistic parallel query processing system providing extensive experimental analysis performed up to 32 processors to show that proposed schemes are superior to the state-of-the-art alternatives.

In Chapter 5, we proposed a query-log-aware replicated partitioning scheme

for the Twitter application. The proposed replicated partitioning scheme models the multi-way relations between social network user actions via a novel temporal activity hypergraph and then utilizing the rpPaToh tool we developed in [5], decides on the distribution of user data. We compare the results of our replicated partitioning scheme with circular replication schemes used in distributed hash tables. The obtained results indicate that the proposed scheme can greatly increase locality.

Though replicated clustering and replicated declustering seem antagonistic problems, in modern data serving facilities where datacenter-, rack- and node-level data placement problems arise, clustering the data in the datacenter- and rack-levels and declustering the data in the node-level is appropriate. Since our log-aware FM-like iterative improvement solutions for the replicated clustering and replicated declustering problems are based from the same roots, we believe they can be used in harmony under such a setup. As future work, we plan to consider the benefits of utilizing our log-aware FM-like iterative improvement solutions over the full datacenter stack.

Bibliography

[1] U. Hoelzle and L. A. Barroso, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st ed., 2009.

[2] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. of the 19th ACM/IEEE Design Automation Conference*, pp. 175–181, 1982.

[3] M. Koyuturk and C. Aykanat, "Iterative-improvement-based declustering heuristics for multi-disk databases," *Information Systems*, vol. 30, pp. 47–70, 2005.

[4] D. R. Liu and M. Y. Wu, "A hypergraph based approach to declustering problems," *Distributed and Parallel Databases*, vol. 10(3), pp. 269–288, 2001.

[5] R. Oguz Selvitopi, A. Turk, and C. Aykanat, "Replicated partitioning for undirected hypergraphs," *J. Parallel Distrib. Comput.*, vol. 72, pp. 547–563, Apr. 2012.

[6] D. R. Liu and S. Shekhar, "Partitioning similarity graphs: a framework for declustering problems," *Information Systems*, vol. 21, pp. 475–496, 1996.

[7] A. S. Tosun, "Threshold-based declustering," *Information Sciences*, vol. 177(5), pp. 1309–1331, 2007.

[8] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 173–182, 1996.

[9] S. Ghemawat, H. Gobio, and S. T. Leung, "The google file system," *ACM SIGOPS Operating Systems Review*, vol. 37(5), pp. 29–43, 2003.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazons highly available key-value store.," in *Proc. 21st ACM SIGOPS symposium on Operating systems principles*, pp. 205–220, 2007.

[11] A. S. Tosun and H. Ferhatosmanoglu, "Optimal parallel i/o using replication," in *Proc. International Workshops on Parallel Processing*, pp. 748–753, 2002.

[12] L. T. Chen and D. Rotem, "Optimal response time retrieval of replicated data," in *Proc. 13th ACM Sigact-Sigmod-Sigart symposium on principles of database systems*, pp. 36–44, 1994.

[13] K. B. Frikken, "Optimal distributed declustering using replication," in *Proc. 19th International Conference on Database Theory - ICDT*, pp. 144–157, 2005.

[14] H. Ferhatosmanoglu, A. S. Tosun, G. Canahuate, and A. Ramachandran, "Efficient parallel processing of range queries through replicated declustering," *Distributed and Parallel Databases*, vol. 20(2), pp. 117–147, 2006.

[15] A. S. Tosun, "Multi-site retrieval of declustered data," in *Proc. 28th Int'l Conf. Distributed Computing Systems*, pp. 486–493, 2008.

[16] A. S. Tosun, "Analysis and comparison of replicated declustering schemes," *IEEE Trans. Parallel Distributed Systems*, vol. 18(11), pp. 1587–1591, 2007.

[17] A. S. Tosun, "Divide-and-conquer scheme for strictly optimal retrieval of range queries," *ACM Trans. on Storage*, vol. 5(3), 2009.

[18] P. Sanders, S. Egner, and K. Korst, "Fast concurrent access to parallel disks," in *Proc. 11th ACm-SIAM Symp. Discrete Algorithms*, pp. 849–858, 2000.

[19] A. S. Tosun, "Design theoretic approach to replicated declustering," in *Proc. Int'l Conf. Information Technology Coding and Computing*, pp. 226–231, 2005.

[20] A. S. Tosun, "Replicated declustering for arbitrary queries," in *Proc. 19th ACM Symp. Applied Computing*, pp. 748–753, 2004.

[21] K. Y. Oktay, A. Turk, and C. Aykanat, "Selective replicated declustering," in *Proc. 15th International Euro-Par Conference on Parallel Processing*, pp. 375–386, 2009.

[22] T. Kwon and S. Lee, "Load-balanced data placement for variable-rate continuous media retrieval," in *Multimedia Database Systems*, pp. 185–207, 1996.

[23] H. Pang, B. Jose, and M. S. Krishnan, "Resource scheduling in a high-performance multimedia server," *IEEE Trans. on Knowl. and Data Eng.*, vol. 11, no. 2, pp. 303–320, 1999.

[24] C. E. Jacobs, A. Finkelstein, and D. H. Salesin, "Fast multiresolution image querying," in *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pp. 277–286, 1995.

[25] L.-T. Liu, M.-T. Kuo, S.-C. Huang, and C.-K. Cheng, "A gradient method on the initial partition of fiduccia-mattheyses algorithm," in *IEEE/ACM international conference on Computer-aided design*, pp. 229–234, 1995.

[26] "Massachussets institute of technology image database." `ftp://whitechapel.media.mit.edu/pub/images/faceimages.zip`, 2005.

[27] E. F. Clark, "Peipa, the pilot european image processing archive." `http://peipa.essex.ac.uk/index.html`.

[28] "The orl database of faces." `http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html`.

[29] H. A. Guvenir and I. Uysal, "Bilkent university function approximation repository." `http://funapp.cs.bilkent.edu.tr`, 2000.

[30] S. Hannan, *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[31] "National transportation atlas databases." CD-ROM, 1999. Bureau of Transportation Statistics.

[32] J. Rao, C. Zhang, N. Megiddo, and G. Lohman, "Automating physical database design in a parallel database," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, (New York, NY, USA), pp. 558–569, ACM, 2002.

[33] R. Arnan, E. Bachmat, T. K. Lam, and R. Michel, "Dynamic data reallocation in disk arrays," *ACM Transaction on Storage*, vol. 3, March 2007.

[34] A. Turk, K. Oktay, and C. Aykanat, "Query-log aware replicated declustering," *Parallel and Distributed Systems, IEEE Transactions on*, 2012.

[35] A. V. Goldberg, "An efficient implementation of a scaling minimum-cost flow algorithm," *J. Algorithms*, vol. 22, no. 1, pp. 1–29, 1997.

[36] H.-I. Hsiao and D. J. DeWitt, "Chained declustering: A new availability strategy for multiprocessor database machines," in *Proceedings of the Sixth International Conference on Data Engineering*, (Washington, DC, USA), pp. 456–465, IEEE Computer Society, 1990.

[37] W. Lang, J. M. Patel, and J. F. Naughton, "On energy management, load balancing and replication," *SIGMOD Rec.*, vol. 38, pp. 35–42, June 2010.

[38] R. Vingralek, Y. Breitbart, and G. Weikum, "Distributed file organization with scalable cost/performance," *SIGMOD Rec.*, vol. 23, pp. 253–264, May 1994.

[39] C. Lu, G. A. Alvarez, and J. Wilkes, "Aqueduct: Online data migration with performance guarantees," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, (Berkeley, CA, USA), USENIX Association, 2002.

[40] J. Tjioe, R. Widjaja, A. Lee, and T. Xie, "Dora: A dynamic file assignment strategy with replication," in *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, (Washington, DC, USA), pp. 148–155, IEEE Computer Society, 2009.

[41] T. Kucukyilmaz, A. Turk, and C. Aykanat, "A parallel framework for in-memory construction of term-partitioned inverted indexes," *The Computer*, 2012.

[42] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval*. New York, NY, USA: McGraw-Hill, Inc., 1986.

[43] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes : Compressing and Indexing Documents and Images*. San Francisco, CA: Morgan Kaufmann, 2. ed., 1999.

[44] A. Tomasic, H. García-Molina, and K. Shoens, "Incremental updates of inverted lists for text document retrieval," in *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, SIGMOD '94, (New York, NY, USA), pp. 289–300, ACM, 1994.

[45] J. Zobel, A. Moffat, and R. Sacks-Davis, "An efficient indexing technique for full text databases," in *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, (San Francisco, CA, USA), pp. 352–362, Morgan Kaufmann Publishers Inc., 1992.

[46] D. W. Harman, "An experimental study of factors important in document ranking," in *Proceedings of the 9th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '86, (New York, NY, USA), pp. 186–193, ACM, 1986.

[47] B. B. Cambazoglu and C. Aykanat, "Performance of query processing implementations in ranking-based text retrieval systems using inverted indices," *Inf. Process. Manage.*, vol. 42, pp. 875–898, July 2006.

[48] V. N. Anh and A. Moffat, "Pruned query evaluation using pre-computed impacts," in *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '06, (New York, NY, USA), pp. 372–379, ACM, 2006.

[49] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. pp. 416–429, 1969.

[50] S. Ding, S. Gollapudi, S. Ieong, K. Kenthapadi, and A. Ntoulas, "Indexing strategies for graceful degradation of search quality," in *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, SIGIR '11, (New York, NY, USA), pp. 575–584, ACM, 2011.

[51] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[52] A. Moffat, W. Webber, and J. Zobel, "Load balancing for term-distributed parallel retrieval," in *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '06, (New York, NY, USA), pp. 348–355, ACM, 2006.

[53] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri, "Mining query logs to optimize index partitioning in parallel web search engines," in *Proceedings of the 2nd international conference on Scalable information systems*, InfoScale '07, (ICST, Brussels, Belgium, Belgium), pp. 43:1–43:9, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007.

[54] R. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations* (R. Miller and J. Thatcher, eds.), pp. 85–103, Plenum Press, 1972.

[55] D. S. Johnson, "Approximation algorithms for combinatorial problems," in *Proceedings of the fifth annual ACM symposium on Theory of computing*, STOC '73, (New York, NY, USA), pp. 38–49, ACM, 1973.

[56] G. Pass, A. Chowdhury, and C. Torgeson, "A picture of search," in *Proceedings of the 1st international conference on Scalable information systems*, InfoScale '06, (New York, NY, USA), ACM, 2006.

[57] Z. Lu and K. S. McKinley, "Partial collection replication versus caching for information retrieval systems," in *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '00, (New York, NY, USA), pp. 248–255, ACM, 2000.

[58] F. Cacheda, V. Plachouras, and I. Ounis, "Performance analysis of distributed architectures to index one terabyte of text," in *Proc. 26th European Conference on IR Research, volume 2997 of Lecture Notes in Computer Science*, pp. 394–408, Springer, 2004.

[59] L. A. Barroso, J. Dean, and U. Hölzle, "Web search for a planet: The google cluster architecture," *IEEE Micro*, vol. 23, pp. 22–28, March 2003.

[60] J. M. Pujol, V. Erramilli, and P. Rodriguez, "Divide and conquer: Partitioning online social networks," *arXiv*, vol. cs.NI, Jan 2009.

[61] J. M. Pujol, G. Siganos, V. Erramilli, and P. Rodriguez, "Scaling online social networks without pains," *Proc of NETDB*, 2009.

[62] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, "The little engine (s) that could: scaling online social networks," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 375–386, 2010.

[63] B. Carrasco, Y. Lu, and J. M. F. da Trindade, "Partitioning social networks for time-dependent queries," in *Proceedings of the 4th Workshop on Social Network Systems*, SNS '11, (New York, NY, USA), pp. 2:1–2:6, ACM, 2011.

[64] C. Berge, *Graphs and Hypergraphs*. North-Holland Publishing Company, 1973.

[65] T. Lengauer, *Combinatorial algorithms for integrated circuit layout*. John Wiley & Sons, Inc., 1990.

[66] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning," in *Proc. of the 36th annual ACM/IEEE Design Automation Conference*, pp. 343–348, 1999.

[67] U. V. Çatalyürek and C. Aykanat, "PaToH: a multilevel hypergraph partitioning tool, version 3.0," tech. rep., Bilkent University, Dept. of Comp. Eng., 1999.

[68] R. Oguz Selvitopi, A. Turk, and C. Aykanat, "rppatoh: Replicated partitioning tool for hypergraphs," Tech. Rep. BU-CE-1203, Bilkent Univ. Comp. Eng. Dept., 2012.

[69] E. Florenzano, "Twitter example for cassandra." `https://github.com/twissandra/twissandra`.

[70] M. De Choudhury, Y.-R. Lin, H. Sundaram, K. S. Candan, L. Xie, and A. Kelliher, "How Does the Data Sampling Strategy Impact the Discovery of Information Diffusion in Social Media?," in *Proceedings of the 4th International AAAI Conference on Weblogs and Social Media*, 2010.

[71] R. Hecht and S. Jablonski, "Nosql evaluation: A use case oriented survey," in *Cloud and Service Computing (CSC), 2011 International Conference on*, pp. 336 –341, dec. 2011.

[72] J. Han, E. Haihong, G. Le, and J. Du, "Survey on nosql database," in *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on*, pp. 363 –366, oct. 2011.

[73] "Project voldemort." `http://project-voldemort.com`, 2012.

[74] "Redis." `http://redis.io`, 2012.

[75] "Scalaris." `http://code.google.com/p/scalaris/`, 2012.

[76] "Membase." `http://www.couchbase.com/membase`, 2012.

[77] "Couchdb." `http://couchdb.apache.org/`, 2012.

[78] "Mongodb." `http://www.mongodb.org/`, 2012.

[79] "Terrastore." `http://code.google.com/p/terrastore`, 2012.

[80] "Riak." `http://basho.com/products/riak-overview/`, 2012.

[81] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, (Berkeley, CA, USA), pp. 15–15, USENIX Association, 2006.

[82] "Hbase." `http://hbase.apache.org/`, 2012.

[83] "Hypertable." `http://hypertable.org/`, 2012.

[84] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, Apr. 2010.

[85] "Neo4j." `http://Neo4j.org/`, 2012.

[86] "Graphdb." `http://www.sones.com/`, 2012.

[87] "Sesame." `http://www.openrdf.org/`, 2012.

[88] "Bigdata." `http://www.systap.com/bigdata.htm`, 2012.

[89] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *Proc. VLDB Endow.*, vol. 3, pp. 48–57, Sept. 2010.